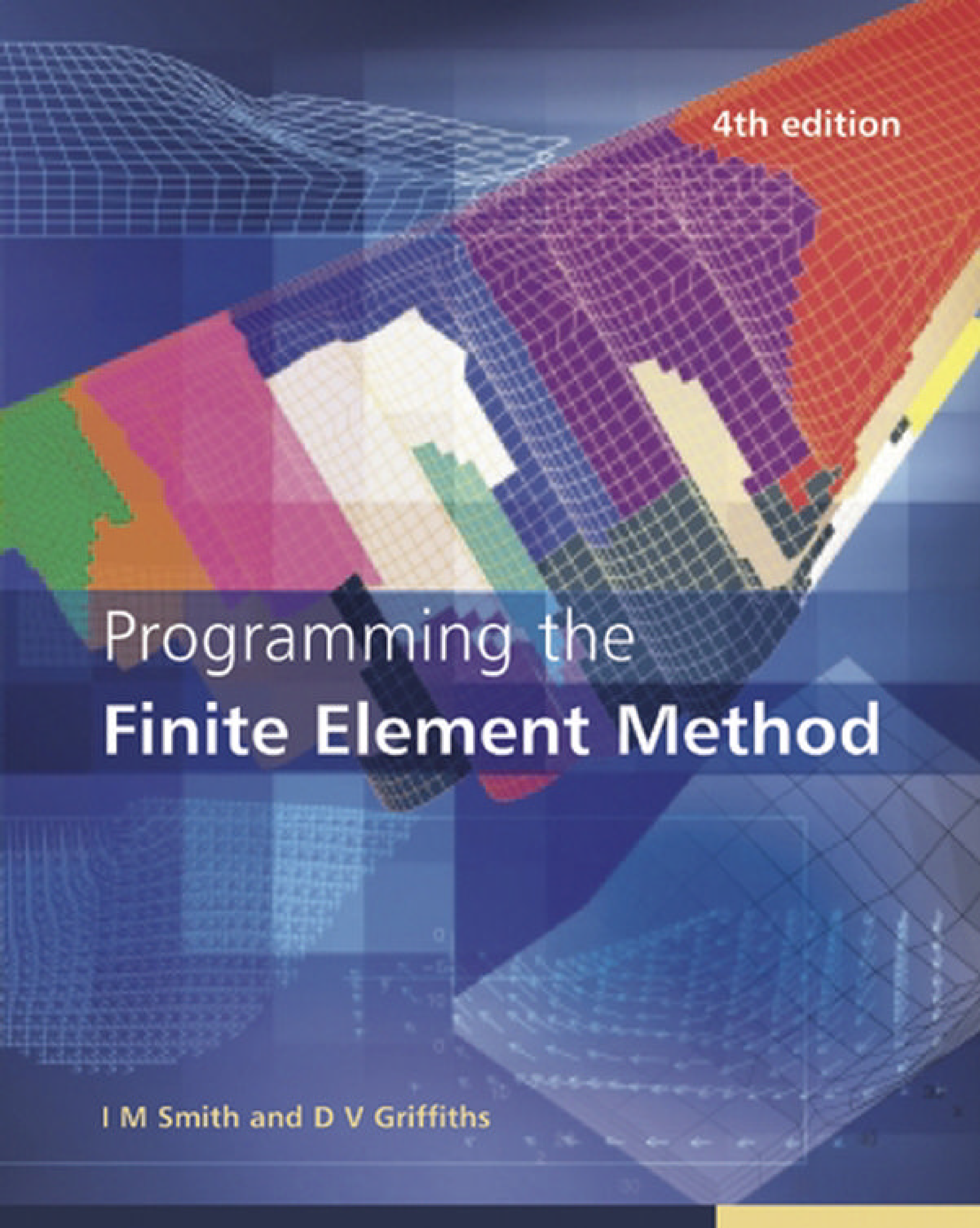


4th edition

Programming the Finite Element Method

I M Smith and D V Griffiths

WILEY

Programming the Finite Element Method

FOURTH EDITION

I. M. Smith
University of Manchester, UK

D. V. Griffiths
Colorado School of Mines, USA

John Wiley & Sons, Ltd

Programming the Finite Element Method

FOURTH EDITION

Programming the Finite Element Method

FOURTH EDITION

I. M. Smith
University of Manchester, UK

D. V. Griffiths
Colorado School of Mines, USA

John Wiley & Sons, Ltd

Library of Congress Cataloging-in-Publication Data

Smith, I. M. (Ian Moffat), 1940-
Programming the finite element method / I.M. Smith, D.V. Griffiths.–4th ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-470-84969-X (alk. paper)—ISBN 0-470-84970-3 (pbk. : alk. paper)
1. Finite element method–Data processing. 2. Engineering–Data Processing. I. Griffiths, D. V. II. Title.

TA347.F5S64 2004
620′.001′51825–dc22

2004048076

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0-470-84969-X (Cloth)
ISBN 0-470-84970-3 (Paper)

Produced from LaTeX files supplied by the authors, typeset by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by TJ International, Padstow, Cornwall
This book is printed on acid-free paper responsibly manufactured from sustainable forestry
in which at least two trees are planted for each one used for paper production.

Contents

Preface to Fourth Edition

The theme of the successful earlier editions has been maintained. A modular programming style, now expressed in Fortran95, facilitates both the consolidation of previous programs into easier-to-use units, and the very important advance into parallel computing environments.

Chapter 1 has been extended by a description of the parallelisation strategy adopted. This involves message passing using the de facto standard, MPI, although other possibilities such as OpenMP are addressed.

In Chapter 3, iterative equation solution methods have been extended to cover non-symmetric systems by the use of bi-conjugate gradient (BiCG) methods. An element-by-element strategy therefore allows the solution of all large finite element problems on modestly sized computers.

From Chapter 4 onwards, programs that were distinct in earlier editions have been consolidated into easier-to-use units within which, for example, problem dimensionality, element type, and so on are selected by the user from inside the same program. Output has been made more concise and readable.

Although mesh generation, a topic in itself, has been avoided, programs that interface readily with mesh generation packages have been provided. For the first time results display using PostScript files has been included in most chapters.

Exercises for students to attempt on their own have been extended.

The major new feature occupies Chapter 12. Therein example programs from Chapters 5 to 11 have been parallelised using element-by-element strategies and MPI. The parallelisation process is largely hidden from the user and the central theme of program conciseness and readability has been maintained. Performance statistics show that efficient use has been made of parallel hardwares ranging from supercomputers to clusters of PCs.

Acknowledgement

Many present and former colleagues have contributed to the present volume but particular thanks are due to D Kidger, L Margetts, F Molenkamp, S Muelas and M Pettipher.

1

Preliminaries: Computer Strategies

1.1 Introduction

Many textbooks exist which describe the principles of the finite element method of analysis and the wide scope of its applications to the solution of practical engineering problems. Usually, little attention is devoted to the construction of the computer programs by which the numerical results are actually produced. It is presumed that readers have access to pre-written programs (perhaps to rather complicated "packages") or can write their own. However, the gulf between understanding in principle what to do, and actually doing it, can still be large for those without years of experience in this field.

The present book bridges this gulf. Its intention is to help readers assemble their own computer programs to solve particular engineering problems by using a "building block" strategy specifically designed for computations via the finite element technique. At the heart of what will be described is not a "program" or a set of programs but rather a collection (library) of procedures or subroutines which perform certain functions analogous to the standard functions (`SIN`, `SQRT`, `ABS`, etc.) provided in permanent library form in all useful scientific computer languages. Because of the matrix structure of finite element formulations, most of the building block routines are concerned with manipulation of matrices.

The building blocks are then assembled in different patterns to make test programs for solving a variety of problems in engineering and science. The intention is that one of these test programs then serves as a platform from which new applications programs are developed by interested users.

The aim of the present book is to teach the reader to write intelligible programs and to use them. Both serial and parallel computing environments are addressed and the building block routines (numbering over 70) and all test programs (numbering over 50) have been verified on a wide range of computers. Efficiency is considered.

The chosen programming language is the latest dialect of FORTRAN, called Fortran 95. Later in this Chapter, a fairly full description of the features of Fortran 95 which influence the programming of the finite element method will be given. At present, all that need be said is that Fortran 95 represents a very radical improvement compared with the previous standard, FORTRAN 77 (which was used in earlier editions of this book), and that Fortran remains, overwhelmingly, the most popular language for writing large engineering and scientific programs. For parallel environments MPI has been used, although the programming strategy has been tested successfully in OpenMP as well.

1.2 Hardware

In principle, any computing machine capable of compiling and running Fortran programs can execute the finite element analyses described in this book. In practice, hardware will range from personal computers for more modest analyses and teaching purposes to "super" computers, usually with parallel processing capabilities, for very large (especially non-linear 3D) analyses. It is a powerful feature of the programming strategy proposed that the same software will run on all machine ranges. The special features of vector and parallel processors are described later (see Sections 1.4 and 1.5).

The user's choice of hardware is a matter of accessibility and of cost. Thus a job taking five minutes on one computer may take one hour on another. Which hardware is "better" clearly depends on individual circumstances. The main advice that can be tendered is against using hardware that is too weak for the task; that is the user is advised not to operate at the extremes of the hardware's capability. If this is done turn round times become too long to be of value in any design cycle. For example, in "virtual prototyping" implementations, execution time has currently to be of the order of 0.1 s to enable refresh graphics to be carried out.

1.3 Memory management

In the programs in this book it will be assumed that sufficient main random access memory (RAM) is available for the storage of data and the execution of programs. However, the arrays processed in finite element calculations might be of size, say, 100,000 by 1000. Thus a computer would need to have a main memory of 10^8 words to hold this information, and while some such computers exist, they are still comparatively rare. A more typical memory size is still of the order of 10^7 words.

One strategy to get round this problem is for the programmer to write "out-of-memory" routines which arrange for the processing of chunks of arrays in memory and the transfer of the appropriate chunks to and from back-up storage.

Alternatively store management is removed from the user's control and given to the system hardware and software. The programmer sees only a single level of memory of very large capacity and information is moved from secondary memory to main memory and out again by the supervisor or executive program which schedules the flow of work through the machine. This concept, namely of a very large "virtual" memory, was first introduced on the ICL ATLAS in 1961, and is now almost universal.

Clearly it is necessary for the system to be able to translate the virtual address of variables into a real address in memory. This translation usually involves a complicated bit-pattern matching called *paging*. The virtual store is split into segments or pages of fixed or variable size referenced by page tables, and the supervisor program tries to "learn" from the way in which the user accesses data in order to manage the store in a predictive way. However, memory management can never be totally removed from the user's control. It must always be assumed that the programmer is acting in a reasonably logical manner, accessing array elements in sequence (by rows or columns as organised by the compiler and the language). If the user accesses a virtual memory of 10^8 words in a random fashion the paging requests will ensure that very little execution of the program can take place (see e.g. Willé, 1995).

In the immediate future, "large" finite element analyses, say involving more than 1 million unknowns, are likely to be processed by the vector and parallel processing hardware described in the next sections. When using such hardware there is usually a considerable time penalty if the programmer interrupts the flow of the computation to perform out-of-memory transfers or if automatic paging occurs. Therefore, in Chapter 3 of this book, special strategies are described whereby large analyses can still be processed "in-memory". However, as problem sizes increase, there is always the risk that main memory, or fast subsidiary memory ("cache") will be exceeded with consequent deterioration of performance on most machine architectures.

1.4 Vector processors

Early digital computers performed calculations "serially", that is, if a thousand operations were to be carried out, the second could not be initiated until the first had been completed, and so on. When operations are being carried out on arrays of numbers, however, it is perfectly possible to imagine that computations in which the result of an operation on two array elements has no effect on an operation on another two array elements, can be carried out simultaneously. The hardware feature by means of which this is realised in a computer is called a *pipeline*, and in general, all modern computers use this feature to a greater or lesser degree. Computers which consist of specialised hardware for pipelining are called *vector* computers. The "pipelines" are of limited length and so for operations to be carried out simultaneously it must be arranged that the relevant operands are actually in the pipeline at the right time. Furthermore, the condition that one operation does not depend on another must be respected. These two requirements (amongst others) mean that some care must be taken in writing programs so that best use is made of the vector processing capacity of many machines. It is moreover an interesting side effect that programs well structured for vector machines will tend to run better on any machine because information tends to be in the right place at the right time (e.g. in a special cache memory) and modern so-called *scalar* computers tend to contain some vector-type hardware. In this book, beginning at Chapter 5, programs which "vectorise" well will be illustrated.

True vector hardware tends to be expensive and at the time of writing a much more common way of increasing processing speed is to execute programs in parallel on many processors. The motivation here is that the individual processors are then "standard" and therefore cheap. However for really intensive computations, it is likely that an amalgamation of vector and parallel hardware is ideal.

1.5 Parallel processors

In this concept (of which there are many variants) there are several physically distinct processors (e.g. a few expensive ones or a lot of cheaper ones). Programs and/or data can reside on different processors which have to communicate with one another.

There are two foreseeable ways in which this communication can be organised (rather like memory management which was described earlier). Either the programmer takes control of the communication process, using a programming feature called *message passing*, or it is done automatically, without user control. The second strategy is of course appealing and has led to the development of "High Performance Fortran" or HPF (e.g. see Koelbel *et al.*, 1995) which has been designed as an extension to Fortran 95. "Directives", which are treated as comments by non-HPF compilers, are inserted into the Fortran 95 programs and allow data to be mapped onto parallel processors together with the specification of the operations on such data which can be carried out in parallel. The attractive feature of this strategy is that programs are "portable", that is they can be easily transferred from computer to computer. One would also anticipate that manufacturers could produce compilers which made best use of their specific type of hardware. At the time of writing, the first implementations of HPF are just being reported.

An alternative to HPF, involving roughly the same level of user intervention, can be used on specific hardware. Manufacturers provide "directives" which can be inserted by users in programs and implemented by the compiler to parallelise sections of the code (usually associated with `DO`-loops). Smith (2000) shows that this approach can be quite effective for up to a modest number of parallel processors (say 10). However such programs are not portable to other machines.

A further alternative is to use OpenMP, a portable set of directives but limited to a class of parallel machines with so-called "shared memory". Although the codes in this book have been rather successfully adapted for parallel processing using OpenMP (Pettipher and Smith, 1997) the most popular strategy applicable equally to "shared memory" and "distributed memory" systems is described in Chapter 12. The programs therein have been run successfully on clusters of PCs communicating via Ethernet and on shared and distributed memory supercomputers with their much more expensive communication systems. This strategy of message passing under programmer control is realised by MPI ("message passing interface") which is a *de facto* standard thereby ensuring portability (MPI Web reference, 2003).

1.6 BLAS libraries

As was mentioned earlier, programs implementing the Finite Element Method make intensive use of matrix or array structures. For example a study of any of the programs in the succeeding chapters will reveal repeated use of the subroutine `MATMUL` described in

Section 1.9. While one might hope that the writers of compilers would implement calls to `MATMUL` efficiently, this turns out in practice not always to be so.

Particularly on supercomputers, an alternative is to use "BLAS" or Basic Linear Algebra Subroutine Libraries (e.g. Dongarra and Walker, 1995). There are three "levels" of BLAS subroutines involving vector—vector, matrix—vector and matrix—matrix operations respectively. To improve efficiency in large calculations, it is always worth experimenting with BLAS routines if available. The calling sequence is rather cumbersome, for example the Fortran:

```
utemp=MATMUL(km,pmul)
```

has to be replaced by:

```
CALL DGEMV('n',ntot,ntot,1.0,km,ntot,pmul,1,0.0,utemp,1)
```

in a typical example in Chapter 12. However, very significant gains in processing speed can be achieved; a factor of 3 times speedup is not uncommon.

1.7 MPI libraries

MPI (MPI Web reference, 2003) is itself essentially a library of routines for communication callable from Fortran. For example,

```
CALL  MPI_BCAST(no_f,fixed_freedoms,MPI_INTEGER,npes-1,MPI_COMM_WORLD,ier)
```

"broadcasts" the array `no_f` of size `fixed_freedoms` to the remaining `npes-1` processors on a parallel system. In the parallel programs in this book (Chapter 12) these MPI routines are mainly hidden from the user and contained within routines collected in library modules such as `gather_scatter`. In this way, the parallel programs can be seen to be readily derived from their serial counterparts. The detail of the new MPI library is left to Chapter 12.

1.8 Applications software

Since all computers have different hardware (instruction formats, vector capability, etc.) and different store management strategies, programs which would make the most effective use of these varying facilities would of course differ in structure from machine to machine. However, for excellent reasons of program portability and programmer training, engineering computations on all machines are usually programmed in "high level" languages which are intended to be machine-independent. The high level language is translated into the machine order code by a program called a *compiler*. Fortran is by far the most widely used language for programming engineering and scientific calculations and in this section the principal features of the latest standard, called *Fortran 95*, will be described with particular reference to features of the language which are useful in finite element computations.

Figure 1.1 shows a typical simple program written in Fortran 95 (Smith, 1995). It concerns an opinion poll survey and serves to illustrate the basic structure of the language for those used to its predecessor, FORTRAN 77, or to other languages.

```
PROGRAM gallup_poll
! TO CONDUCT A GALLUP POLL SURVEY
 IMPLICIT NONE
 INTEGER::sample,i,count,this_time,last_time,tot_rep,tot_mav,tot_dem,  &
   tot_other,rep_to_mav,dem_to_mav,changed_mind
 READ*,sample
 count=0; tot_rep=0; tot_mav=0; tot_dem=0; tot_other=0; rep_to_mav=0
 dem_to_mav=0; changed_mind=0
 OPEN(10,FILE='gallup.dat')
 DO I=1,sample
   count=count+1
   READ(10,'(I3,I2)',ADVANCE='NO')this_time,last_time
   votes: SELECT CASE(this_time)
   CASE(1); tot_rep=tot_rep+1
   CASE(3); tot_mav=tot_mav+1
     IF(last_time/=3)THEN
       changed_mind=changed_mind+1
       IF(last_time==1)rep_to_mav=rep_to_mav+1
       IF(last_time==2)dem_to_mav=dem_to_mav+1
     END IF
   CASE(2); tot_dem=tot_dem+1
   CASE DEFAULT; tot_other=tot_other+1
   END SELECT votes
 END DO
 PRINT*,'PERCENT REPUBLICAN IS', REAL (tot_rep)/REAL (count)*100.0
 PRINT*,'PERCENT MAVERICK IS', REAL (tot_mav)/REAL(count)*100.0
 PRINT*,'PERCENT DEMOCRAT IS', REAL (tot_mav)/REAL(count)*100.0
 PRINT*,'PERCENT OTHERS IS', REAL (tot_other)/REAL(count)*100.0
 PRINT*,'PERCENT CHANGING REP TO MAVIS',                              &
   REAL (rep_to_mav)/REAL(changed_mind)*100.0
 PRINT*,'PERCENT CHANGING DEM TO MAV   IS',                           &
   REAL (dem_to_mav)/REAL(changed_mind)*100.0
STOP
END PROGRAM gallup_poll
```

Figure 1.1 A typical program written in Fortran 95

It can be seen that programs are written in "free source" form. That is, statements can be arranged on the page or screen at the user's discretion. Other features to note are:

- Upper and lower case characters may be mixed at will. In the present book, upper case is used to signify intrinsic routines and "key words" of Fortran 95.
- Multiple statements can be placed on one line, separated by `;`.
- Long lines can be extended by `&` at the end of the line, and optionally another `&` at the start of the continuation line(s).
- Comments placed after `!` are ignored.
- Long names (up to 31 characters, including the underscore) allow meaningful identifiers.
- The `IMPLICIT NONE` statement forces the declaration of all variable and constant names. This is of great help in debugging programs.
- Declarations involve the `::` double colon convention.
- There are no labelled statements.

1.8.1 Arithmetic

Finite element processing is computationally intensive (see e.g. Chapters 6 and 10) and a reasonably safe numerical precision to aim for is that provided by a 64-bit machine word length. Fortran 95 contains some useful intrinsic procedures for determining, and changing, processor precision. For example the statement

```
iwp = SELECTED_REAL_KIND(15)
```

would return an integer `iwp`, which is the `KIND` of variable on a particular processor which is necessary to achieve 15 decimal places of precision. If the processor cannot achieve this order of accuracy, `iwp` would be returned as negative.

Having established the necessary value of `iwp`, Fortran 95 declarations of `REAL` quantities then take the form

```
REAL(iwp)::a,b,c
```

and assignments the form

```
a=1.0_iwp; b=2.0_iwp; c=3.0_iwp
```

and so on.

In most of the programs in this book, constants are assigned at the time of declaration, for example,

```
REAL(iwp)::zero=0.0_iwp,one=1.0_iwp,d4=4.0_iwp,penalty=1.0e20_iwp
```

so that the rather cumbersome `_iwp` extension does not appear in the main program assignment statements.

1.8.2 Conditions

There are two basic structures for conditional statements in Fortran 95 which are both shown in Figure 1.1. The first corresponds to the classical `IF ... THEN ... ELSE` structure found in most high level languages. It can take the form:

```
 name_of_clause: IF(logical expression 1)THEN
   .  first block
   .  of statements
   .
   ELSE IF(logical expression 2)THEN
   .  second block
   .  of statements
   .
   ELSE
   .  third block
   .  of statements
   .
END IF name_of_clause
```

For example,

```
change_sign: IF(a/=b)THEN
  a=-a
ELSE
  b=-b
END IF change_sign
```

The name of the conditional statement, `name_of_clause` or `change_sign` in the above examples, is optional and can be left out.

The second conditional structure involves the `SELECT CASE` construct. If choices are to be made in particularly simple circumstances, for example, an `INTEGER`, `LOGICAL` or `CHARACTER` scalar has a given value then the form:

```
select_case_name: SELECT CASE(variable or expression)
CASE(selector)
  .     first block
  .     of statements
  .
CASE(selector)
  .     second block
  .     of statements
  .
CASE DEFAULT
  .     default block
  .     of statements
  .
END select_case_name
```

can be used. This replaces the ugly "computed go to" construct in FORTRAN 77.

1.8.3 Loops

There are two constructs in Fortran 95 for repeating blocks of instructions. In the first, the block is repeated a fixed number of times, for example

```
fixed_iterations: DO i=1,n
  .     block
  .     of statements
  .
END DO fixed_iterations
```

In the second, the loop is left or continued depending on the result of some condition. For example

```
exit_type: DO
  .     block
  .     of statements
  .
  IF(conditional statement)EXIT
  .     block
```

```
   .     of statements
   .
 END DO exit_type
```

or

```
 cycle_type: DO
   .     block
   .     of statements
   .
   IF(conditional statement)CYCLE
   .     block
   .     of statements
   .
 END DO cycle_type
```

The first variant transfers control out of the loop to the first statement after `END DO`. The second variant transfers control to the beginning of the loop, skipping the remaining statements between `CYCLE` and `END DO`.

In the above examples, as was the case for conditions, the naming of the loops is optional. In the programs in this book, loops and conditions of major significance tend to be named and simpler ones not.

1.9 Array features

1.9.1 Dynamic arrays

Fortran 95 has remedied perhaps the greatest deficiency of earlier FORTRANs for large scale array computations such as occur in finite element analysis, in that it allows “dynamic” declaration of arrays. That is, array sizes do not have to be specified at program compilation time but can be `ALLOCATEd` after some data has been read into the program, or some intermediate results computed. A simple illustration is given below:

```
PROGRAM dynamic
!  just to illustrate dynamic array allocation
 IMPLICIT NONE
 iwp=SELECTED_REAL_KIND(15)
!  declare variable space for two-dimensional array a
 REAL,ALLOCATABLE(iwp)::a(:,:)
 REAL::two=2.0_iwp,d3=3.0_iwp
 INTEGER::m,n
!  now read in the bounds for a
 READ*,m,n
!  allocate actual space for a
 ALLOCATE(a(m,n))
 READ*,a
 PRINT*,two*SQRT(a)+d3
 DEALLOCATE(a)! a no longer needed
STOP
END PROGRAM dynamic
```

This simple program also illustrates some other very useful features of the standard. "Whole array" operations are permissible, so that the whole of an array is read in, or the square root of all its elements computed, by a single statement. The efficiency with which these features are implemented by practical compilers is variable.

1.9.2 Broadcasting

A feature called *broadcasting* enables operations on whole arrays by scalars such as `two` or `d3` in the above example. These scalars are said to be "broadcast" to all the elements of the array so that what will be printed out are the square roots of all the elements of the array having been multiplied by 2.0 and added to 3.0.

1.9.3 Constructors

Array elements can be assigned values in the normal way but Fortran 95 also permits the "construction" of one-dimensional arrays, or vectors, such as the following:

```
v = (/1.0,2.0,3.0,4.0,5.0/)
```

which is equivalent to

```
v(1)=1.0; v(2)=2.0; v(3)=3.0; v(4)=4.0; v(5)=5.0
```

Array constructors can themselves be arrays, for example

```
w = (/v, v/)
```

would have the obvious result for the 10 numbers in `w`.

1.9.4 Vector subscripts

Integer vectors can be used to define subscripts of arrays, and this is very useful in the "gather" and "scatter" operations involved in finite element (and other numerical) methods. Figure 1.2 shows a portion of a finite element mesh of 8-node quadrilaterals with its nodes numbered "globally" at least up to 106 in the example shown. When "local" calculations have to be done involving individual elements, for example to determine element strains or fluxes, a local index vector could hold the node numbers of each element, that is:

82	76	71	72	73	77	84	83	for element 65
93	87	82	83	84	88	95	94	for element 73

and so on. This index or "steering" vector could be called `g`. When a local vector has to be gathered from a global one,

```
local = global(g)
```

is valid, and for scattering,

```
global(g) = local
```

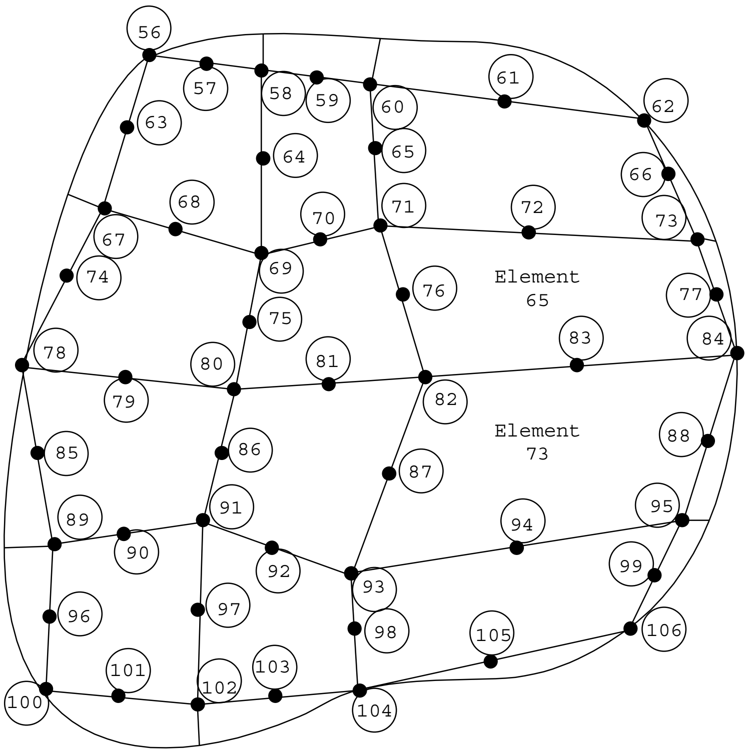


Figure 1.2 Portion of a finite element mesh with node and element numbers

In this example `local` and `g` would be 8-long vectors, whereas `global` could have a length of thousands or millions.

1.9.5 Array sections

Parts of arrays or "subarrays" can be referenced by giving an integer range for one or more of their subscripts. If the range is missing for any subscript, the whole extent of that dimension is implied. Thus if `a` and `b` are two-dimensional arrays, `a(:,1:3)` and `b(11:13,:)` refer to all the terms in the first three columns of a, and all the terms in rows 11 through 13 of `b` respectively. If array sections "conform", that is, have the right number of rows and columns, they can be manipulated just like "whole" arrays.

1.9.6 Whole-array manipulations

It is worth emphasising that the array-computation features in Fortran 95 remove the need for several subroutines which were essential in FORTRAN 77 and used in earlier editions of this book. For example, if a, b, and c conform and s is a scalar, the following are valid:

```
 a = b+c   !   no need for a matrix add, involving DO loops
 a = b-c   !   no need for a matrix subtract
 a = s*b   !   no need for a matrix-scalar multiply by s
 a = b/s   !   no need for a matrix-scalar divide by s
 a = 0.0   !   no need for a matrix null
```

However, although `a = b*c` has a meaning for conforming arrays a, b, and c, its consequence is the computation of the element-by-element products of b and c and is not to be confused with the matrix multiply described in the next sub-section.

1.9.7 Intrinsic procedures for arrays

To supplement whole-array arithmetic operations, Fortran 95 provides a few intrinsic procedures (functions) which are very useful in finite element work. These can be grouped conveniently into those involving array computations, and those involving array inspection. The array computation functions are

```
 FUNCTION MATMUL(a,b)           !   returns matrix product of a and b
 FUNCTION DOT_PRODUCT(v1,v2)    !   returns dot product of v1 and v2
 FUNCTION TRANSPOSE(a)          !   returns transpose of a.
```

All three are heavily used in the programs in this book and replace the user-written subroutines which had to be provided in previous FORTRAN 77 editions.

The array inspection functions include:

```
FUNCTION MAXVAL(a)     ! returns the element of an array a of
                       ! maximum value (not absolute maximum)
FUNCTION MINVAL(a)     ! returns the element of an array a of
                       ! minimum value (not absolute minimum)
FUNCTION MAXLOC(a)     ! returns the location of the maximum element
                       ! of array a
FUNCTION MINLOC(a)     ! returns the location of the minimum element
                       ! of array a
FUNCTION PRODUCT(a)    ! returns the product of all the elements of a
FUNCTION SUM(a)        ! returns the sum of all the elements of a
FUNCTION LBOUND(a,1)   ! returns the first lower bound of a, etc.
FUNCTION UBOUND(a,1)   ! returns the first upper bound of a, etc.
```

The first six of these procedures allow an optional argument called a *masking* argument. For example the statement

```
 asum=SUM(column,MASK=column>=0.0)
```

will result in asum containing the sum of the positive elements of array column.

Useful procedures whose only argument is a MASK are:

```
ALL(MASK=column>=0.0)   ! true if all elements of column are positive
ANY(MASK=column>=0.0)   ! true if any elements of column are positive
COUNT(MASK=column<=0.0)! number of elements of column which are
                        ! negative.
```

For multidimensional arrays, operations such as SUM can be carried out on a particular dimension of the array. When a mask is used, the dimension argument must be specified even if the array is one-dimensional. Referring to Figure 1.2, the “half-bandwidth” of a particular element could be found from the element freedom steering vectors, g, by the statement

```
nband = MAXVAL(g,1,g>0) - MINVAL(g,1,g>0)
```

allowing for the possibility of zero entries in g. Note that the argument MASK= is optional.

The global “half-bandwidth” of an assembled system of equation coefficients would then be the maximum value of nband after scanning all the elements in the mesh.

1.9.8 Additional Fortran 95 features

The programs in this book are written in a style of Fortran 95 not too far removed from that of FORTRAN 77. The examples of FORTRAN 77 and Fortran 95 shown in Figure 1.3, illustrate gains in conciseness from whole array operations, array intrinsic functions, and dynamic arrays, but no complete revolution in programming style has been implemented.

Fortran 95 contains features such as derived data types, pointers, operator overloading, and user-defined operators which programmers used to another style might implement to bring about a more radical revision of FORTRAN 77. This is a matter of taste. One feature of Fortran 95 which has been implemented in the programs which follow is the idea of a “module”.

A module is a program unit separate from the main program unit in the way that subroutines and functions are. However, in its simplest form, it may contain no executable

Fortran 95

```
 km=zero
 int_pts_1: DO i=1,nip
   CALL shape_der(der,points,i); jac=MATMUL(der,coord)
   det=determinant(jac); CALL invert(jac)
   deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
   km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
 END DO int_pts_1
```

FORTRAN 77

```
      CALL NULL(KM,IKM,IDOF,IDOF)
      DO 20 I=1,NIP
      CALL FORMLN(DER,IDER,FUN,SAMP,ISAMP,I)
      CALL MATMUL(DER,IDER,COORD,ICOORD,JAC,IJAC,IT,NOD,IT)
      CALL TWOBYTWO (JAC,IJAC,JAC1,IJAC1,DET)
      CALL MATMUL(JAC1,IJAC1,DER,IDER,DERIV,IDERIV,IT,IT,NOD)
      CALL FORMB(BEE,IBEE,DERIV,IDERIV,NOD)
      CALL MATMUL(DEE,IDEE,BEE,IBEE,DBEE,IDBEE,IH,IH,IDOF)
      CALL MATRAN(BT,IBT,BEE,IBEE,IH,IDOF)
      CALL MATMUL(BT,IBT,DBEE,IDBEE,BTDB,IBTDB,IDOF,IH,IDOF)
      QUOT=DET*WEIGHTS(I)
      CALL MSMULT(BTDB,IBTDB,QUOT,IDOF,IDOF)
   20 CALL MATADD(KM,IKM,BTDB,IBTDB,IDOF,IDOF)
```

Figure 1.3 Comparison of a portion of a finite element program in Fortran 95 with FORTRAN 77

statements at all and just be a list or collection of declarations or data which is globally accessible to the program unit which invokes it by a `USE` statement. Its main employment later in the book will be to contain either a collection of subroutines and functions which constitute a "library" or to contain the "interfaces" between such a library and a program which uses it.

1.9.9 Subprogram libraries

It was stated in the Introduction to this Chapter that what will be presented in Chapter 4 onwards is not a monolithic program but rather a collection of test programs which all access a common subroutine library which contains about 70 subroutines and functions. In the simplest implementation of Fortran 95 the library routines could simply be appended to the main program after a `CONTAINS` statement as follows:

```
PROGRAM test_one
     .
     .
     .
     .
CONTAINS
SUBROUTINE one(p1,p2,p3)
     .
     .
     .
END SUBROUTINE one
SUBROUTINE two(p4,p5,p6)
     .
     .
     .
END SUBROUTINE two
     .
   etc.
END PROGRAM test_one
```

This would be tedious because a sub-library would really be required for each test program, containing only the needed subroutines. Secondly, compilation of the library routines with each test program compilation is wasteful.

What is required, therefore, is for the whole subroutine library to be precompiled and for the test programs to link only to the parts of the library which are needed.

The designers of Fortran 95 seem to have intended this to be done in the following way. The subroutines would be placed in a file:

```
SUBROUTINE one(args1)
      .
      .
      .
END SUBROUTINE one
SUBROUTINE two(args2)
      .
      .
    etc.
SUBROUTINE ninety_nine(args99)
      .
```

```
     .
     .
END SUBROUTINE ninety_nine
```

and compiled.

A “module” would constitute the interface between library and calling program. It would take the form

```
MODULE main
  INTERFACE
    SUBROUTINE one(args1)
       (Parameter declarations)
    END SUBROUTINE one
    SUBROUTINE two(args2)
       (Parameter declarations)
        .
        .
       etc.
    SUBROUTINE ninety_nine(args99)
       (Parameter declarations)
    END SUBROUTINE ninety_nine
  END INTERFACE
END MODULE main
```

Thus the interface module would contain only the subroutine “headers”, that is the subroutine’s name, argument list, and declaration of argument types. This is deemed to be safe because the compiler can check the number and type of arguments in each call (one of the greatest sources of error in FORTRAN 77).

The libraries would be interfaced by a statement `USE main` at the beginning of each test program. For example

```
PROGRAM test_program1
  USE main
      .
      .
      .
END PROGRAM test_program1
```

However, it is still quite tedious to keep updating two files when making changes to a library (the library and the interface module). Users with straightforward Fortran 95 libraries may well prefer to omit the interface stage altogether and just create a module containing the subroutines themselves. These would then be accessed by `USE library_routines` in the example shown below. This still allows the compiler to check the numbers and types of subroutine arguments when the test programs are compiled. For example

```
MODULE library_routines
  CONTAINS
  SUBROUTINE one(args1)
       .
       .
       .
  END SUBROUTINE one
  SUBROUTINE two(args2)
       .
       .
      etc.
```

```
   SUBROUTINE ninety_nine(args99)
        .
        .
        .
   END SUBROUTINE ninety_nine
 END MODULE library_routines
```

and then

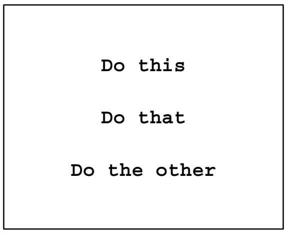
```
PROGRAM test_program_2
 USE library_routines
         .
         .
         .
END PROGRAM test_program_2
```

1.9.10 Structured programming

The finite element programs which will be described are strongly "structured" in the sense of Dijkstra (1976). The main feature exhibited by our programs will be seen to be a nested structure and we will use representations called "structure charts" (Lindsey, 1977) rather than flow charts to describe their actions.

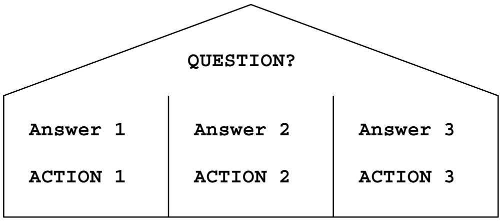
The main features of these charts are:

(i) The block



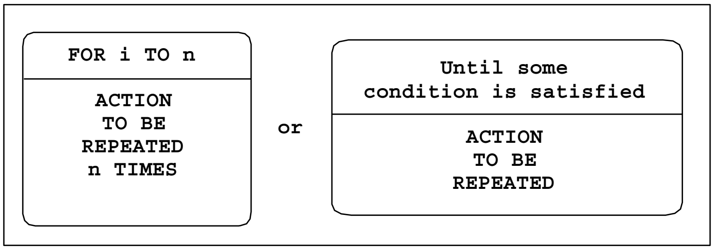
This will be used for the outermost level of each structure chart. Within a block, the indicated actions are to be performed sequentially.

(ii) The choice



This corresponds to the `IF...THEN...ELSE IF...THEN....END IF` or `SELECT CASE` type of construct.

(iii) The loop


FOR i TO n
ACTION TO BE REPEATED n TIMES
or
Until some condition is satisfied
ACTION TO BE REPEATED

This comes in various forms, but we shall usually be concerned with DO-loops, either for a fixed number of repetitions or "forever" (so called because of the danger of the loop never being completed).

In particular, the structure chart notation discourages the use of GOTO statements. Using this notation, a matrix multiplication program would be represented as shown in Figure 1.4. The nested nature of a typical program can be seen quite clearly.

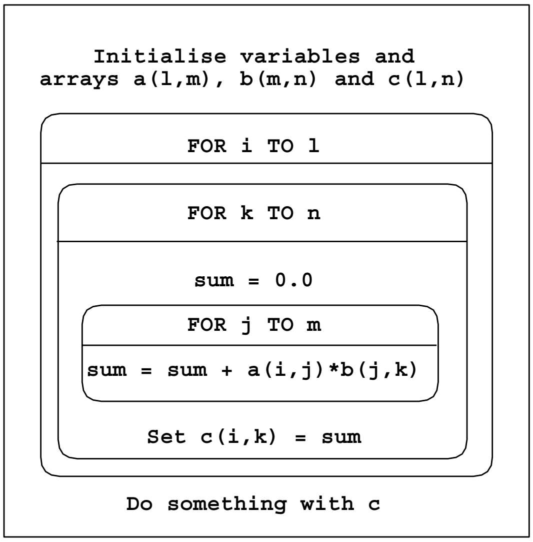

Initialise variables and arrays a(l,m), b(m,n) and c(l,n)
FOR i TO l
FOR k TO n
sum = 0.0
FOR j TO m
sum = sum + a(i,j)*b(j,k)
Set c(i,k) = sum
Do something with c

Figure 1.4 Structure chart for matrix multiplication

1.10 Conclusions

Computers on which finite element computations can be done vary widely in their capabilities and architecture. Because of its entrenched position FORTRAN is the language in which computer programs for engineering applications had best be written in order to assure maximum readership and portability. Using Fortran 95, a library of subroutines can be created which is held in compiled form and accessed by programs in just the way that a manufacturer's permanent library is. For parallel implementations a similar strategy is adopted using MPI. Further information on parallel implementations is at `www.parafem.org.uk`.

Using this philosophy, a library of over 70 subroutines has been assembled, together with some 50 example programs which access the library. These programs and subroutines are written in a reasonably "structured" style, and can be downloaded from the Internet at `www.mines.edu/fs_home/vgriffit/4th_ed`. Versions are at present available for all the common machine ranges and Fortran 95 compilers. The downloadable programs include the MPI library, which consists of only some 12 subroutines, and the 10 example programs from Chapter 12 which use them.

The structure of the remainder of the book is as follows. Chapter 2 shows how the differential equations governing the behaviour of solids and fluids are semi-discretised in space using finite elements.

Chapter 3 describes the subprogram library and the basic techniques by which main programs are constructed to solve the equations listed in Chapter 2. Two basic solution strategies are described, one involving element matrix assembly to form global matrices, which can be used for small to medium-sized problems and the other using "element-by-element" matrix techniques to avoid assembly and therefore permit the solution of very large problems.

The remaining Chapters 4 to 12 are concerned with applications, partly in the authors' field of geomechanics. However, the methods and programs described are equally applicable in many other fields of engineering and science such as structural mechanics, fluid dynamics, electromagnetics and so on. Chapter 4 leads off with static analysis of skeletal structures. Chapter 5 deals with static analysis of linear solids, while Chapter 6 discusses extensions to deal with material non-linearity. Programs dealing with the common geotechnical process of construction (element addition during the analysis) and excavation (element removal during the analysis) are given. Chapter 7 is concerned with steady state problems (e.g. fluid or heat flow) while transient states with inclusion of transport phenomena (diffusion with advection) are treated in Chapter 8. In Chapter 9, coupling between solid and fluid phases is treated, with applications to "consolidation" processes in geomechanics. A second type of "coupling" which is treated involves the Navier–Stokes equations. Chapter 10 contains programs for the solution of eigenvalue problems (e.g. steady state vibration), involving the determination of natural modes by various methods. Integration of the equations of motion in time is described in Chapter 11. Chapter 12 takes 10 example programs from earlier chapters and shows how these may be parallelised using the MPI library. Since only "large" problems benefit from parallelisation, all of these examples employ three-dimensional geometries.

In every applications chapter, test programs are listed and described, together with specimen input and output. At the conclusion of most chapters, exercise questions are included, with solutions.

References

Dijkstra EW 1976 *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J.

Dongarra JJ and Walker DW 1995 Software libraries for linear algebra computations on high-performance computers. *SIAM Rev* **37**(2), 151–180.

Koelbel CH, Loveman DB, Schreiber RS, Steele GL and Zosel ME 1995 *The High Performance Fortran Handbook*. MIT Press, Cambridge, Mass.

Lindsey CH 1977 Structure charts: a structured alternative to flow charts. *SIGPLAN Notices* **12**(11), 36–49.

MPI Web Reference 2003 `http://www-unix.mcs.anl.gov/mpi/`.

Pettipher MA and Smith IM 1997 The development of an MPP implementation of a suite of finite element codes. *High-Performance Computing and Networking: Lecture Notes in Computer Science*. Springer-Verlag, Berlin, pp. 1225:400–409.

Smith IM 1995 *Programming in Fortran 90*. John Wiley & Sons, Chichester, New York.

Smith IM 2000 A general-purpose system for finite element analyses in parallel. *Eng Comput* **17**(1), 75–91.

Willé DR 1995 *Advanced Scientific Fortran*. John Wiley & Sons, Chichester, New York.

2

Spatial Discretisation by Finite Elements

2.1 Introduction

The finite element method is a technique for solving partial differential equations by first discretising these equations in their space dimensions. The discretisation is carried out locally over small regions of simple but arbitrary shape (the finite elements). This results in matrix equations relating the input at specified points in the elements (the nodes) to the output at these same points. In order to solve equations over large regions, the matrix equations for the smaller sub-regions can be summed node by node, resulting in global matrix equations, or "element-by-element" techniques can be employed to avoid creating (large) global matrices. The method is already described in many texts, for example, Zienkiewicz and Taylor (1989), Strang and Fix (1973), Cook *et al.* (1989), and Rao (1989), but the principles will briefly be described in this chapter in order to establish a notation and to set the scene for the later descriptions of programming techniques.

2.2 Rod element

2.2.1 Rod stiffness matrix

Figure 2.1(a) shows the simplest solid element, namely an elastic rod, with end nodes 1 and 2. The element has length L while u denotes the longitudinal displacements of points on the rod which is subjected to axial loading only.

If P is the axial force in the rod at a particular section and F is an applied body force (units of force/length) then,

$$P = \sigma A = EA\varepsilon = EA\frac{\mathrm{d}u}{\mathrm{d}x} \tag{2.1}$$

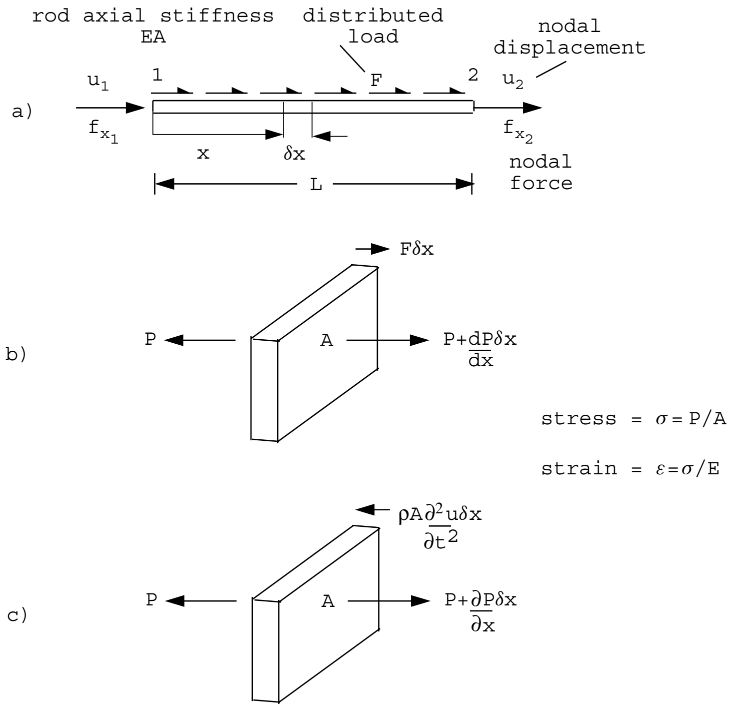


Figure 2.1 Equilibrium of a rod element

assuming "small" strain, and for equilibrium from Figure 2.1(b),

$$\frac{\mathrm{d}P}{\mathrm{d}x} + F = 0 \tag{2.2}$$

hence the differential equation to be solved is

$$EA\frac{\mathrm{d}^2u}{\mathrm{d}x^2} + F = 0 \tag{2.3}$$

In the finite element technique, the continuous variable u is approximated by $\tilde{u}$ in terms of its nodal values, u_1 and u_2, through simple functions of the space variable called *shape functions*. That is

$$\tilde{u} = N_1u_1 + N_2u_2$$

or

$$\tilde{u} = [N_1 \;\; N_2]\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = [\mathbf{N}]\{\mathbf{u}\} \tag{2.4}$$

where

$$N_1 = 1 - \frac{x}{L},\;\; N_2 = \frac{x}{L} \tag{2.5}$$

If the true variation in u is higher order, as will often be the case, greater accuracy could be achieved by introducing higher order shape functions or by including more linear subdivisions.

When (2.4) is substituted in (2.3), we have

$$EA\frac{\mathrm{d}^2}{\mathrm{d}x^2}[N_1 \quad N_2]\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} + F = \mathcal{R} \tag{2.6}$$

where $\mathcal{R}$ is a measure of the error in the approximation and is called the *residual*. The differential equation has thus been replaced by an equation in terms of the nodal values u_1 and u_2. The problem now reduces to one of finding "good" values for u_1 and u_2 in order to minimise the residual $\mathcal{R}$.

Many methods could be used to achieve this. For example Griffiths and Smith (1991) discuss collocation, subdomain, Galerkin, and least squares techniques. Of these, Galerkin's method, for example Finlayson (1972), is the most widely used in finite element work. The method consists of multiplying or "weighting" the residual in (2.6) by each shape function in turn, integrating over the element and equating to zero. Thus

$$\int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} EA\frac{\mathrm{d}^2}{\mathrm{d}x^2}[N_1 \quad N_2]\,\mathrm{d}x \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} + \int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} F\,\mathrm{d}x = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \tag{2.7}$$

Note that in the present example, in which the shape functions are linear, double differentiation of these functions would cause them to vanish. This difficulty is resolved by applying Green's theorem (integration by parts) to yield typically

$$\int N_i \frac{\partial^2 N_j}{\partial x^2}\,\mathrm{d}x = -\int \frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x}\,\mathrm{d}x + \text{boundary terms, which we usually ignore} \tag{2.8}$$

Hence, assuming EA and F are not functions of x, (2.7) becomes

$$-EA\int_0^L \begin{bmatrix} \frac{\partial N_1}{\partial x}\frac{\partial N_1}{\partial x} & \frac{\partial N_1}{\partial x}\frac{\partial N_2}{\partial x} \\ \frac{\partial N_2}{\partial x}\frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x}\frac{\partial N_2}{\partial x} \end{bmatrix} \mathrm{d}x \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} + F\int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} \mathrm{d}x = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \tag{2.9}$$

On evaluation of the integrals,

$$-EA\begin{bmatrix} \frac{1}{L} & -\frac{1}{L} \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} + F\begin{Bmatrix} \frac{L}{2} \\ \frac{L}{2} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \tag{2.10}$$

The above case is for a uniformly distributed force F acting along the element, and it should be noted that the Galerkin procedure has resulted in the total force FL being shared equally between the two nodes. If in Figure 2.1(a) the loading is applied only at the nodes we have

$$\frac{EA}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = \begin{Bmatrix} f_{x_1} \\ f_{x_2} \end{Bmatrix} \tag{2.11}$$

where f_{x_1} is the force in the x-direction at node 1 etc. Equation (2.11) represents the rod element stiffness relationship, which in matrix notation becomes,

$$[\mathbf{k}_m]\{\mathbf{u}\} = \{\mathbf{f}\} \tag{2.12}$$

where $[\mathbf{k}_m]$ is the "element stiffness matrix", $\{\mathbf{u}\}$ is the element nodal "displacements vector", and $\{\mathbf{f}\}$ is the element nodal "forces vector".

2.2.2 Rod mass element

Consider now the case of an unrestrained rod in free longitudinal vibration. Figure 2.1(c) shows the equilibrium of a segment in which the body force is now given by Newton's law as mass times acceleration. If the mass per unit volume is ρ, the partial differential equation becomes,

$$EA\frac{\partial^2 u}{\partial x^2} - \rho A\frac{\partial^2 u}{\partial t^2} = 0 \tag{2.13}$$

On discretising u in space by finite elements as before, the first term in (2.13) clearly leads again to $[\mathbf{k}_m]$. The second term takes the form

$$-\int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} \rho A\,[N_1 \quad N_2]\,\mathrm{d}x\frac{\mathrm{d}^2}{\mathrm{d}t^2}\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} \tag{2.14}$$

and assuming that ρA is not a function of x,

$$-\rho A\int_0^L \begin{bmatrix} N_1N_1 & N_1N_2 \\ N_2N_1 & N_2N_2 \end{bmatrix}\mathrm{d}x\frac{\mathrm{d}^2}{\mathrm{d}t^2}\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} \tag{2.15}$$

Evaluation of integrals yields

$$-\frac{\rho AL}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}\frac{\mathrm{d}^2}{\mathrm{d}t^2}\begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} \tag{2.16}$$

or in matrix notation

$$-[\mathbf{m}_m]\left\{\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}t^2}\right\}$$

where $[\mathbf{m}_m]$ is the "element mass matrix". Thus the full matrix statement of equation (2.13) is

$$[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{m}_m]\left\{\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}t^2}\right\} = \{\mathbf{0}\} \tag{2.17}$$

which is a set of ordinary differential equations.

Note that $[\mathbf{m}_m]$ formed in this manner is the "consistent" mass matrix and differs from the "lumped" equivalent which would lead to $\rho AL/2$ terms on the diagonal with zeros off-diagonal.

2.3 The eigenvalue equation

Equation (2.17) is sometimes integrated directly (Chapter 11) but is also the starting point for derivation of the eigenvalues or natural frequencies of single elements or meshes of elements.

Suppose the elastic rod element is undergoing free harmonic motion. Then all nodal displacements will be harmonic, of the form,

$$\{\mathbf{u}\} = \{\mathbf{a}\}\sin(\omega t + \psi) \tag{2.18}$$

where $\{\mathbf{a}\}$ are amplitudes of the motion, ω its frequency and ψ its phase shift. When (2.18) is substituted in (2.17), the equation

$$[\mathbf{k}_m]\{\mathbf{a}\} - \omega^2[\mathbf{m}_m]\{\mathbf{a}\} = \{\mathbf{0}\} \tag{2.19}$$

is obtained, which can easily be rearranged as a standard eigenvalue equation. Chapter 10 describes solution of equations of this type.

2.4 Beam element

2.4.1 Beam element stiffness matrix

As a second one-dimensional solid element, consider the slender beam in Figure 2.2. The end nodes 1 and 2 are subjected to shear forces and moments which result in translations and rotations. Each node, therefore, has 2 "degrees of freedom".

The element shown in Figure 2.2 has length L, flexural rigidity EI, and carries a uniform transverse load of q (units of force/length). The well known equilibrium equation for this system is given by,

$$EI\frac{\mathrm{d}^4 w}{\mathrm{d}x^4} = q \tag{2.20}$$

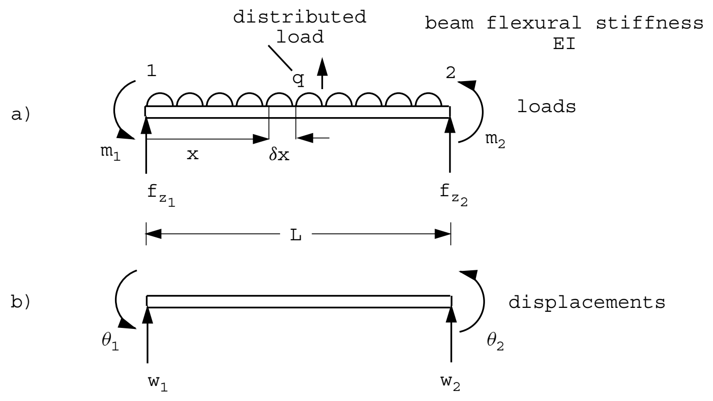


Figure 2.2 Slender beam element

Again the continuous variable, w in this case, is approximated in terms of discrete nodal values, but we introduce the idea that not only w itself but also its derivatives θ can be used in the approximation. In this case the continuous variable w is approximated by $\tilde{w}$ in terms of nodal values as follows:

$$\tilde{w} = [N_1 \quad N_2 \quad N_3 \quad N_4] \begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} = [\mathbf{N}]\{\mathbf{w}\} \tag{2.21}$$

where $\theta_1 = \mathrm{d}w/\mathrm{d}x$ at node 1, and so on. In this case, (2.21) can often be made exact by choosing the cubic shape functions:

$$\begin{aligned} N_1 &= \frac{1}{L^3}(L^3 - 3Lx^2 + 2x^3) \\ N_2 &= \frac{1}{L^2}(L^2x - 2Lx^2 + x^3) \\ N_3 &= \frac{1}{L^3}(3Lx^2 - 2x^3) \\ N_4 &= \frac{1}{L^2}(x^3 - Lx^2) \end{aligned} \tag{2.22}$$

Note that the shape functions have the property that they, or their derivatives in this case, equal one at a specific node and zero at all others.

Substitution in (2.20) and application of Galerkin's method leads to the four element equations:

$$\int_0^L \begin{Bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{Bmatrix} EI \frac{\mathrm{d}^4}{\mathrm{d}x^4} [N_1 \quad N_2 \quad N_3 \quad N_4] \, \mathrm{d}x \begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} = \int_0^L \begin{Bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{Bmatrix} q \, \mathrm{d}x \tag{2.23}$$

Again Green's theorem is used to avoid differentiating four times; for example

$$\int N_i \frac{\mathrm{d}^4 N_j}{\mathrm{d}x^4} \, \mathrm{d}x \approx -\int \frac{\mathrm{d}N_i}{\mathrm{d}x} \frac{\mathrm{d}^3 N_j}{\mathrm{d}x^3} \, \mathrm{d}x \approx \int \frac{\mathrm{d}^2 N_i}{\mathrm{d}x^2} \frac{\mathrm{d}^2 N_j}{\mathrm{d}x^2} \, \mathrm{d}x + \text{neglected terms} \tag{2.24}$$

Hence assuming EI and q are not functions of x, (2.23) becomes

$$EI \int_0^L \left[\frac{\mathrm{d}^2 N_i}{\mathrm{d}x^2} \frac{\mathrm{d}^2 N_j}{\mathrm{d}x^2} \right]_{i,j=1,2,3,4} \mathrm{d}x \begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} = q \int_0^L \begin{Bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{Bmatrix} \mathrm{d}x \tag{2.25}$$

Evaluation of the integrals gives,

$$\frac{2EI}{L^3}\begin{bmatrix} 6 & 3L & -6 & 3L \\ & 2L^2 & -3L & L^2 \\ & & 6 & -3L \\ \text{symmetrical} & & & 2L^2 \end{bmatrix}\begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} = \frac{qL}{12}\begin{Bmatrix} 6 \\ L \\ 6 \\ -L \end{Bmatrix} \tag{2.26}$$

which recovers the standard "slope-deflection" equations for beam elements.

The above case is for a uniformly distributed load applied to the beam. For the case in which loading is applied only at the nodes we have,

$$\frac{2EI}{L^3}\begin{bmatrix} 6 & 3L & -6 & 3L \\ & 2L^2 & -3L & L^2 \\ & & 6 & -3L \\ \text{symmetrical} & & & 2L^2 \end{bmatrix}\begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} = \begin{Bmatrix} f_{z_1} \\ m_1 \\ f_{z_2} \\ m_2 \end{Bmatrix} \tag{2.27}$$

which represents the beam element stiffness relationship.

Hence, in matrix notation we again have,

$$[\mathbf{k}_m]\{\mathbf{w}\} = \{\mathbf{f}\} \tag{2.28}$$

Beam–column elements, in which axial and bending effects are combined from (2.11) and (2.27), are described further in Chapter 4.

2.4.2 Beam element mass matrix

If the element in Figure 2.2 were vibrating transversely, it would be subjected to an additional restoring force $-\rho A(\partial^2 w/\partial t^2)$. The matrix form, by analogy with (2.15), is just,

$$-\rho A\int_0^L \begin{bmatrix} N_1N_1 & N_1N_2 & N_1N_3 & N_1N_4 \\ N_2N_1 & N_2N_2 & N_2N_3 & N_2N_4 \\ N_3N_1 & N_3N_2 & N_3N_3 & N_3N_4 \\ N_4N_1 & N_4N_2 & N_4N_3 & N_4N_4 \end{bmatrix} \mathrm{d}x\frac{\mathrm{d}^2}{\mathrm{d}t^2}\begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} \tag{2.29}$$

and evaluation of the integrals yields the beam element mass matrix given by,

$$[\mathbf{m}_m] = \frac{\rho AL}{420}\begin{bmatrix} 156 & 22L & 54 & -13L \\ & 4L^2 & 13L & -3L^2 \\ & & 156 & -22L \\ \text{symmetrical} & & & 4L^2 \end{bmatrix} \tag{2.30}$$

In this instance, the approximation of the consistent mass terms by lumped ones can lead to large errors in the prediction of beam frequencies as shown by Leckie and Lindberg (1963). Strategies for lumping the mass matrix of a beam element are described further in Chapter 10.

2.5 Beam with an axial force

If the beam element in Figure 2.2 is subjected to an additional axial force P, as shown in Figure 2.3, a simple modification to (2.20) results in the differential equation

$$EI\frac{\mathrm{d}^4w}{\mathrm{d}x^4} \pm P\frac{\mathrm{d}^2w}{\mathrm{d}x^2} = q \tag{2.31}$$

where the positive sign corresponds to a compressive axial load and vice versa.

Finite element discretisation and application of Galerkin's method leads to an additional matrix associated with the axial force contribution,

$$\mp P\int_0^L \left[\frac{\mathrm{d}N_i}{\mathrm{d}x}\frac{\mathrm{d}N_j}{\mathrm{d}x}\right]_{i,j=1,2,3,4} \mathrm{d}x \begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} \tag{2.32}$$

On discretising w in space by finite elements as before, the first term in (2.31) clearly leads again to $[\mathbf{k}_m]$. The second term from (2.32) takes the form for compressive P,

$$P\frac{1}{30L}\begin{bmatrix} 36 & 3L & -36 & 3L \\ & 4L^2 & -3L & -L^2 \\ & & 36 & -3L \\ \text{symmetrical} & & & 4L^2 \end{bmatrix}\begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} \tag{2.33}$$

The matrix is sometimes called the *beam geometric matrix*, since it is a function only of the length of the beam, given by,

$$[\mathbf{k}_g] = \frac{1}{30L}\begin{bmatrix} 36 & 3L & -36 & 3L \\ & 4L^2 & -3L & -L^2 \\ & & 36 & -3L \\ \text{symmetrical} & & & 4L^2 \end{bmatrix} \tag{2.34}$$

and the equilibrium equation can be written as:

$$([\mathbf{k}_m] - P[\mathbf{k}_g])\{\mathbf{w}\} = \{\mathbf{f}\} \tag{2.35}$$

Buckling of a member can be investigated by solving the eigenvalue problem where $\{\mathbf{f}\} = \{\mathbf{0}\}$, or by increasing the compressive force P on the element until large deformations result or in simple cases, by determinant search. Equations (2.34) and (2.35) represent an approximation of the approach to modifying the element stiffness involving stability

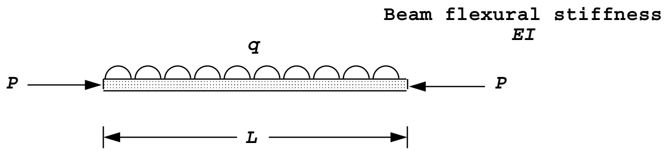


Figure 2.3 Beam with an axial force

functions (Horne and Merchant, 1965). The accuracy of the approximation depends on the value of P/P_E for each member, where P_E is the Euler load. Over the range $-1 < P/P_E < 1$ the approximation introduces errors no greater than 7% (Livesley, 1975). For larger positive values of P/P_E, however, (2.35) can become inaccurate unless more element subdivisions are used. Program 4.6 uses a simple iterative approach to compute the buckling load of beams and beams on elastic foundations.

2.6 Beam on an elastic foundation

In Figure 2.4 a continuous elastic support has been placed beneath the beam element. If this support has stiffness k (units of force/length^2) then clearly the transverse load is resisted by an extra force kw leading to the differential equation,

$$EI\frac{d^4w}{dx^4} + kw = q \tag{2.36}$$

By comparison of the second term with the inertia restoring force $-\rho A \partial^2 w/\partial t^2$ from equation (2.13), it will be apparent that application of the Galerkin process to (2.36) will result in a foundation stiffness matrix that is identical to the consistent mass matrix from (2.30), apart from the coefficient k instead of ρA. The equilibrium equation will then be of the form

$$([\mathbf{k}_m] + [\mathbf{m}_m])\{\mathbf{w}\} = \{\mathbf{f}\} \tag{2.37}$$

A "lumped mass" approach to this problem is also possible by simply adding the appropriate spring stiffness to the diagonal terms of the beam stiffness matrix (see e.g. Griffiths, 1989).

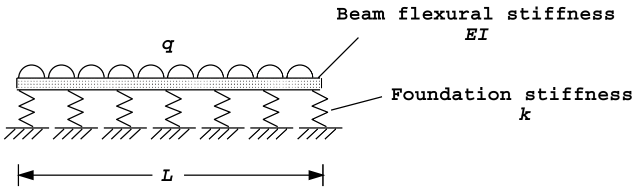


Figure 2.4 Beam on a continuous elastic foundation

2.7 General remarks on the discretisation process

Enough examples have now been described for a general pattern to emerge of how terms in a differential equation appear in matrix form after discretisation. Table 2.1 gives a summary, N_i being the shape functions.

In fact, first order terms such as du/dx have not yet arisen. They are unique in Table 2.1 in leading to matrix equations which are not symmetrical, as indeed would be the case for any odd order of derivative. We shall return to terms of this type in Chapter 8, in relation to advection in fluid flow.

Table 2.1 Semi-discretisation of partial differential equations

Term in differential equation	Typical term in matrix equation	Symmetry?
u	$\int N_i N_j \, \mathrm{d}x$	Yes
$\frac{\mathrm{d}u}{\mathrm{d}x}$	$\int N_i \frac{\mathrm{d}N_j}{\mathrm{d}x} \mathrm{d}x$	No
$\frac{\mathrm{d}^2u}{\mathrm{d}x^2}$	$-\int \frac{\mathrm{d}N_i}{\mathrm{d}x}\frac{\mathrm{d}N_j}{\mathrm{d}x} \mathrm{d}x$	Yes
$\frac{\mathrm{d}^4u}{\mathrm{d}x^4}$	$\int \frac{\mathrm{d}^2N_i}{\mathrm{d}x^2}\frac{\mathrm{d}^2N_j}{\mathrm{d}x^2} \, \mathrm{d}x$	Yes

2.8 Alternative derivation of element stiffness

Instead of working from the governing differential equation, element properties can often be derived by an alternative method based on a consideration of energy. For example, the strain energy stored due to bending of a very small length δx of the elastic beam element in Figure 2.2 is,

$$\delta U = \frac{1}{2}\frac{M^2}{EI}\delta x \tag{2.38}$$

where M is the "bending moment" and by conservation of energy this must be equal to the work done by the external loads q, thus

$$\delta W = \frac{1}{2}qw\delta x \tag{2.39}$$

The bending moment M is related to w through the "moment-curvature" expression,

$$M = -EI\frac{\mathrm{d}^2w}{\mathrm{d}x^2}$$

or

$$M = [\mathbf{D}]\{\mathbf{A}\}w \tag{2.40}$$

where $[\mathbf{D}]$ is the material property EI and $\{\mathbf{A}\}$ is the operator $-\mathrm{d}^2/\mathrm{d}x^2$. Writing (2.38) in the form,

$$\delta U = \frac{1}{2}\left(-\frac{\mathrm{d}^2w}{\mathrm{d}x^2}\right)M\delta x \tag{2.41}$$

we have

$$\delta U = \frac{1}{2}\left(\{\mathbf{A}\}w\right)^{\mathrm{T}}M\delta x \tag{2.42}$$

Introducing the discretised approximation $\tilde{w}$, from (2.21) and (2.40) this becomes

$$\begin{aligned}\delta U &= \frac{1}{2}\left(\{\mathbf{A}\}[\mathbf{N}]\{\mathbf{w}\}\right)^{\mathrm{T}}[\mathbf{D}]\{\mathbf{A}\}[\mathbf{N}]\{\mathbf{w}\}\,\delta x \\ &= \frac{1}{2}\{\mathbf{w}\}^{\mathrm{T}}\left(\{\mathbf{A}\}[\mathbf{N}]\right)^{\mathrm{T}}[\mathbf{D}]\{\mathbf{A}\}[\mathbf{N}]\{\mathbf{w}\}\,\delta x \end{aligned} \tag{2.43}$$

The total strain energy of the element is thus,

$$U = \frac{1}{2}\int_0^L \{\mathbf{w}\}^{\mathrm{T}}\left(\{\mathbf{A}\}[\mathbf{N}]\right)^{\mathrm{T}}[\mathbf{D}]\{\mathbf{A}\}[\mathbf{N}]\{\mathbf{w}\}\,\mathrm{d}x \tag{2.44}$$

The product $\{\mathbf{A}\}[\mathbf{N}]$ is usually written as $[\mathbf{B}]$, and since $\{\mathbf{w}\}$ are nodal values and therefore, constants

$$U = \frac{1}{2}\{\mathbf{w}\}^{\mathrm{T}}\int [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}][\mathbf{B}]\,\mathrm{d}x\,\{\mathbf{w}\} \tag{2.45}$$

Similar operations on (2.39) lead to the total external work done, W, and hence the stored potential energy of the beam is given by

$$\begin{aligned}\Pi &= U - W \\ &= \frac{1}{2}\{\mathbf{w}\}^{\mathrm{T}}\int_0^L [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}][\mathbf{B}]\,\mathrm{d}x\,\{\mathbf{w}\} - \frac{1}{2}\{\mathbf{w}\}^{\mathrm{T}} q\int_0^L [\mathbf{N}]^{\mathrm{T}}\,\mathrm{d}x \end{aligned} \tag{2.46}$$

A state of stable equilibrium is achieved when Π is a minimum with respect to all $\{\mathbf{w}\}$. That is,

$$\frac{\partial \Pi}{\partial\{\mathbf{w}\}^{\mathrm{T}}} = \int_0^L [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}][\mathbf{B}]\,\mathrm{d}x\,\{\mathbf{w}\} - q\int_0^L [\mathbf{N}]^{\mathrm{T}}\,\mathrm{d}x = 0 \tag{2.47}$$

or

$$\int_0^L [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}][\mathbf{B}]\,\mathrm{d}x\,\{\mathbf{w}\} = q\int_0^L [\mathbf{N}]^{\mathrm{T}}\,\mathrm{d}x \tag{2.48}$$

which is simply another way of writing (2.25).

Thus we see from (2.28) that the elastic element stiffness matrix $[\mathbf{k}_m]$ can be written in the form,

$$[\mathbf{k}_m] = \int_0^L [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}][\mathbf{B}]\,\mathrm{d}x \tag{2.49}$$

which will prove to be a useful general matrix form for expressing stiffnesses of all elastic solid elements. The computer programs for analysis of solids developed in the next chapter use this notation and method of stiffness formation.

The "energy" formulation described above is clearly valid only for "conservative" systems. Galerkin's method is more generally applicable.

2.9 Two-dimensional elements: plane strain and plane stress

The elements so far described have not been true finite elements because they have been used to solve differential equations in one space variable only. Thus the real problem involving two or three space variables has been replaced by a hypothetical, equivalent one-dimensional problem before solution. The elements we have considered can be joined together at points (the nodes) and complete continuity (compatibility) and equilibrium achieved. In this way we can sometimes obtain exact solutions to our hypothetical problems (especially at the nodes) in which solutions will be unaffected by the number of elements chosen to represent uniform line segments.

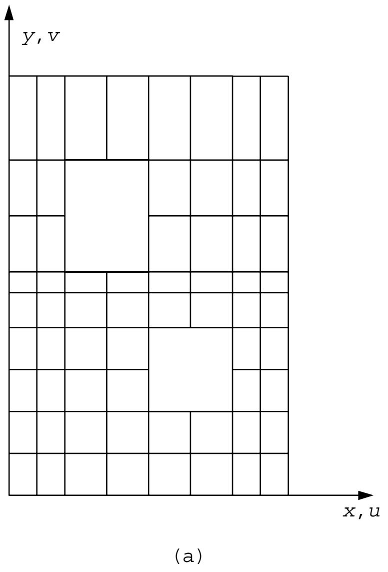
This situation changes radically when problems in two or three space dimensions are analysed. For example, consider the plane shear wall with openings shown in Figure 2.5(a). The wall has been sub-divided into rectangular elements of side lengths a and b of which Figure 2.5(b) is typical. These elements have 4 corner nodes so that when the idealised wall is assembled, the elements will only be attached at these points.

If the wall can be considered to be of unit thickness and in a state of plane stress, (Timoshenko and Goodier, 1982), the equations to be solved are the following:

1. Equilibrium

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + F_x = 0$$

$$\frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + F_y = 0 \qquad (2.50)$$



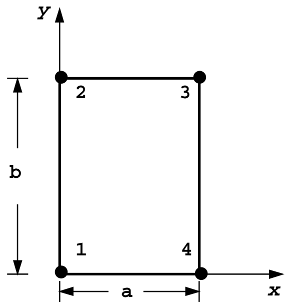


Figure 2.5 (a) Shear wall with openings. (b) Typical rectangular 4-node element

where σ_x, σ_y and τ_{xy} are the only non-zero stress components and F_x, F_y are "body forces" (units of force/length^3).

2. Constitutive (plane stress)

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{Bmatrix} \tag{2.51}$$

where E is Young's modulus, ν is Poisson's ratio, and ϵ_x, ϵ_y and γ_{xy} are the independent small strain components.

3. Strain-displacement

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{Bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix} \tag{2.52}$$

where u and v are the components of displacement in the x and y directions.

Equations (2.50) to (2.52) can be written in the form

$$\begin{aligned} [\mathbf{A}]^{\mathrm{T}} \{\boldsymbol{\sigma}\} &= -\{\mathbf{f}\} \\ \{\boldsymbol{\sigma}\} &= [\mathbf{D}]\{\boldsymbol{\epsilon}\} \\ \{\boldsymbol{\epsilon}\} &= [\mathbf{A}]\{\mathbf{e}\} \end{aligned} \tag{2.53}$$

where

$$\{\boldsymbol{\sigma}\} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix}, \{\boldsymbol{\epsilon}\} = \begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{Bmatrix}, \{\mathbf{e}\} = \begin{Bmatrix} u \\ v \end{Bmatrix}, \quad \{\mathbf{f}\} = \begin{Bmatrix} F_x \\ F_y \end{Bmatrix} \tag{2.54}$$

$$[\mathbf{A}] = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}, \quad [\mathbf{D}] = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{2.55}$$

We shall only be concerned in this book with "displacement" formulations in which $\{\boldsymbol{\sigma}\}$ and $\{\boldsymbol{\epsilon}\}$ are eliminated from (2.53) as follows:

$$\begin{aligned} [\mathbf{A}]^{\mathrm{T}} \{\boldsymbol{\sigma}\} &= -\{\mathbf{f}\} \\ [\mathbf{A}]^{\mathrm{T}} [\mathbf{D}]\{\boldsymbol{\epsilon}\} &= -\{\mathbf{f}\} \\ [\mathbf{A}]^{\mathrm{T}} [\mathbf{D}][\mathbf{A}]\{\mathbf{e}\} &= -\{\mathbf{f}\} \end{aligned} \tag{2.56}$$

Writing out (2.56) in full we have

$$\frac{E}{1-\nu^2}\left\{\begin{matrix} \dfrac{\partial^2 u}{\partial x^2}+\dfrac{1-\nu}{2}\dfrac{\partial^2 u}{\partial y^2}+\nu\,\dfrac{\partial^2 v}{\partial x \partial y}+\dfrac{1-\nu}{2}\dfrac{\partial^2 v}{\partial y \partial x} \\ \nu\,\dfrac{\partial^2 u}{\partial y \partial x}+\dfrac{1-\nu}{2}\dfrac{\partial^2 u}{\partial x \partial y}+\dfrac{1-\nu}{2}\dfrac{\partial^2 v}{\partial x^2}+\dfrac{\partial^2 v}{\partial y^2} \end{matrix}\right\}=\left\{\begin{matrix} -F_x \\ -F_y \end{matrix}\right\} \tag{2.57}$$

which is a pair of simultaneous partial differential equations in the continuous space variables u and v.

As usual these can be solved by discretising over each element using shape functions (here we assume the same functions in the x- and y-directions)

$$\tilde{u}=[N_1 \;\; N_2 \;\; N_3 \;\; N_4]\left\{\begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{matrix}\right\}=[\mathbf{N}]\{\mathbf{u}\} \tag{2.58}$$

and

$$\tilde{v}=[N_1 \;\; N_2 \;\; N_3 \;\; N_4]\left\{\begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix}\right\}=[\mathbf{N}]\{\mathbf{v}\} \tag{2.59}$$

where in the case of the 4-node rectangular element shown in Figure 2.5(b) the N_i functions were first derived by Taig (1961) to be

$$\begin{aligned} N_1 &= \left(1-\frac{x}{a}\right)\left(1-\frac{y}{b}\right) \\ N_2 &= \left(1-\frac{x}{a}\right)\frac{y}{b} \\ N_3 &= \frac{x}{a}\frac{y}{b} \\ N_4 &= \frac{x}{a}\left(1-\frac{y}{b}\right) \end{aligned} \tag{2.60}$$

These result in linear variations in strain across the element which is sometimes called the *linear strain rectangle*.

Discretisation and application of Galerkin's method (Szabo and Lee, 1969), using Table 2.1, leads to the stiffness equations for a typical element,

$$\frac{E}{1-\nu^2}\int_0^a\int_0^b \left[\begin{matrix} \left(\frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x}+\frac{1-\nu}{2}\frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial y}\right) & \left(\nu\frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial y}+\frac{1-\nu}{2}\frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial x}\right) \\ \left(\nu\frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial x}+\frac{1-\nu}{2}\frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial y}\right) & \left(\frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial y}+\frac{1-\nu}{2}\frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x}\right) \end{matrix}\right]_{i,j=1,2,3,4} \mathrm{d}x\,\mathrm{d}y\,\{\mathbf{u}\}=\{\mathbf{f}\} \tag{2.61}$$

where $\{\mathbf{u}\}$ and $\{\mathbf{f}\}$ are the nodal displacements and force components. Most programs in this book arrange these components by alternating them, thus $\{\mathbf{u}\}=[u_1\ v_1\ u_2\ v_2\ u_3\ v_3\ u_4\ v_4]^{\mathrm{T}}$

and $\{\mathbf{f}\} = [f_{x1}\ f_{y1}\ f_{x2}\ f_{y2}\ f_{x3}\ f_{y3}\ f_{x4}\ f_{y4}]^{\mathrm{T}}$ where u_1 is the x-displacement at node 1, and f_{y2} is the y-force at node 2, etc.

The stiffness relationship can also be written in the standard form of equation (2.28) as,

$$[\mathbf{k}_m]\{\mathbf{u}\} = \{\mathbf{f}\} \tag{2.62}$$

Evaluation of the first term in the plane stress stiffness matrix yields,

$$k_{m1,1} = \frac{E}{1-\nu^2}\left(\frac{b}{3a} + \frac{1-\nu}{2}\frac{a}{3b}\right) \tag{2.63}$$

and so on.

Note that the size of the element does not appear in this expression, only the ratio a/b (or b/a), which is called the *aspect ratio* of the element.

Integration by parts of the weighted form of (2.61) now leads to integrals of the type,

$$\iint N_i \frac{\partial^2 N_j}{\partial x^2}\,\mathrm{d}x\,\mathrm{d}y = -\iint \frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x}\,\mathrm{d}x\,\mathrm{d}y + \int_S N_i \frac{\partial N_j}{\partial x}\, l_n\,\mathrm{d}S \tag{2.64}$$

where l_n is the direction cosine of the normal to boundary S and we assume that the contour integral in (2.64) is zero between elements. This assumption is generally reasonable but extra care is needed at mesh boundaries. Only if the elements become vanishingly small can our solution be the correct one (an infinite number of elements) except in trivial cases. Physically, in a displacement method, it is usual to satisfy compatibility everywhere in a mesh but to satisfy equilibrium only at the nodes. It is also possible to violate compatibility, but none of the elements described in this book does.

2.10 Energy approach

As was done in the case of the elastic beam element, the principle of minimum potential energy can be used to provide an alternative derivation of (2.62) for elastic plane elements. The element strain energy per unit thickness is

$$\begin{aligned} U &= \iint \frac{1}{2}\{\boldsymbol{\sigma}\}^{\mathrm{T}}\{\boldsymbol{\epsilon}\}\,\mathrm{d}x\,\mathrm{d}y \\ &= \frac{1}{2}\{\mathbf{u}\}^{\mathrm{T}}\iint ([\mathbf{A}][\mathbf{S}])^{\mathrm{T}}\mathbf{D}([\mathbf{A}][\mathbf{S}])\,\mathrm{d}x\,\mathrm{d}y\,\{\mathbf{u}\} \qquad (2.65) \\ &= \frac{1}{2}\{\mathbf{u}\}^{\mathrm{T}}\iint [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}][\mathbf{B}]\,\mathrm{d}x\,\mathrm{d}y\,\{\mathbf{u}\} \qquad (2.66) \end{aligned}$$

where $[\mathbf{A}]$ and $[\mathbf{D}]$ are defined in (2.55),

$$[\mathbf{S}] = \begin{bmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 \end{bmatrix} \tag{2.67}$$

and $[\mathbf{B}] = [\mathbf{A}]\,[\mathbf{S}]$, leading to,

$$[\mathbf{B}] = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \frac{\partial N_3}{\partial x} & 0 & \frac{\partial N_4}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & 0 & \frac{\partial N_3}{\partial y} & 0 & \frac{\partial N_4}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial y} & \frac{\partial N_3}{\partial x} & \frac{\partial N_4}{\partial y} & \frac{\partial N_4}{\partial x} \end{bmatrix} \tag{2.68}$$

Thus, we have again for this element

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}}\,[\mathbf{D}]\,[\mathbf{B}]\;\mathrm{d}x\,\mathrm{d}y \tag{2.69}$$

which is the form in which it will be computed in Chapter 3.

Exactly the same expression holds in the case of plane strain, but the elastic [**D**] matrix becomes (Timoshenko and Goodier, 1982), for unit thickness,

$$[\mathbf{D}] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{2.70}$$

2.11 Plane element mass matrix

When inertia is significant (2.57) are supplemented by forces $-\rho\partial^2 u/\partial t^2$ and $-\rho\partial^2 v/\partial t^2$ where ρ is the mass of the element per unit volume. For an element of unit thickness this leads, in exactly the same way as in (2.15), to the element mass matrix which has terms given by

$$[\mathbf{m}_m] = \rho \iint [\mathbf{N}]^{\mathrm{T}}\,[\mathbf{N}]\;\mathrm{d}x\,\mathrm{d}y \tag{2.71}$$

and hence to an eigenvalue equation the same as (2.19).

Evaluation of the first term in the plane element mass matrix as illustrated in Figure 2.5(b) yields,

$$m_{m1,1} = \frac{\rho ab}{9} \tag{2.72}$$

2.12 Axisymmetric stress and strain

Solids of revolution subjected to axisymmetric loading possess only two independent components of displacement and can be analysed as if they were two-dimensional. For example,

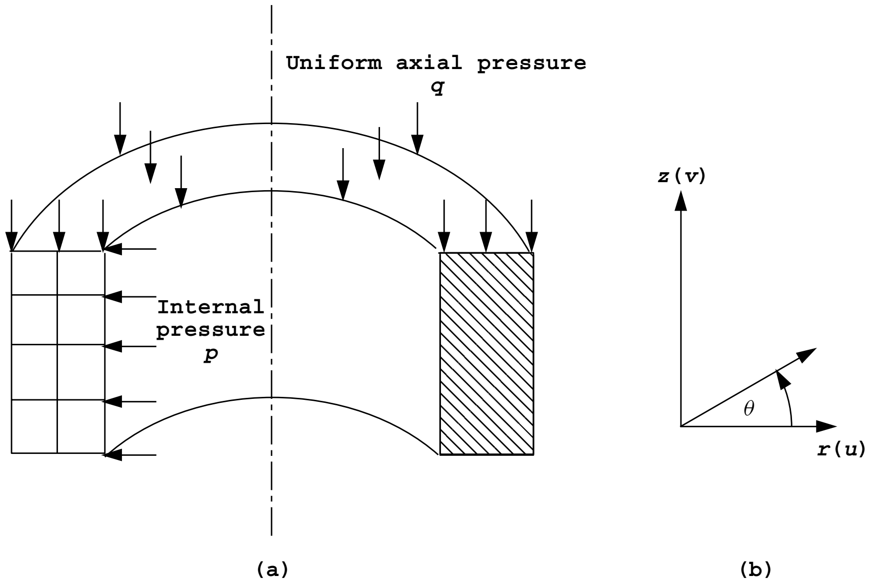


Figure 2.6 (a) Cylinder under axial and radial pressure. (b) Cylindrical coordinate system

Figure 2.6(a) shows a thick tube subjected to radial pressure p and axial pressure q. Only a typical radial cross-section need be analysed and is sub-divided into rectangular elements in the figure. The cylindrical coordinate system, Figure 2.6(b), is the most convenient and when it is used the element stiffness equation equivalent to (2.69) is

$$[\mathbf{k}_m] = \iiint [\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}] \, r \, \mathrm{d}r \, \mathrm{d}z \, \mathrm{d}\theta \tag{2.73}$$

which, when integrated over one radian, becomes

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}] \, r \, \mathrm{d}r \, \mathrm{d}z \tag{2.74}$$

where the strain-displacement relations are now (Timoshenko and Goodier, 1982)

$$\begin{Bmatrix} \epsilon_r \\ \epsilon_z \\ \gamma_{rz} \\ \epsilon_\theta \end{Bmatrix} = \begin{bmatrix} \dfrac{\partial}{\partial r} & 0 \\ 0 & \dfrac{\partial}{\partial z} \\ \dfrac{\partial}{\partial z} & \dfrac{\partial}{\partial r} \\ \dfrac{1}{r} & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix} \tag{2.75}$$

or $\{\boldsymbol{\epsilon}\} = [\mathbf{A}]\{\mathbf{e}\}$, where u and v now represent displacement components in the r and z directions.

As before $[\mathbf{B}] = [\mathbf{A}]\,[\mathbf{S}]$ leading to

$$[\mathbf{B}] = \begin{bmatrix} \frac{\partial N_1}{\partial r} & 0 & \frac{\partial N_2}{\partial r} & 0 & \frac{\partial N_3}{\partial r} & 0 & \frac{\partial N_4}{\partial r} & 0 \\ 0 & \frac{\partial N_1}{\partial z} & 0 & \frac{\partial N_2}{\partial z} & 0 & \frac{\partial N_3}{\partial z} & 0 & \frac{\partial N_4}{\partial z} \\ \frac{\partial N_1}{\partial z} & \frac{\partial N_1}{\partial r} & \frac{\partial N_2}{\partial z} & \frac{\partial N_2}{\partial r} & \frac{\partial N_3}{\partial z} & \frac{\partial N_3}{\partial r} & \frac{\partial N_4}{\partial z} & \frac{\partial N_4}{\partial r} \\ \frac{N_1}{r} & 0 & \frac{N_2}{r} & 0 & \frac{N_3}{r} & 0 & \frac{N_4}{r} & 0 \end{bmatrix} \tag{2.76}$$

where for elements of rectangular cross-section, [**N**] could again be defined by (2.60). In axisymmetric analysis, four independent stress and strain terms must be retained, so the stress-strain matrix is redefined as

$$[\mathbf{D}] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & 1 & 0 & \frac{\nu}{1-\nu} \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 1 \end{bmatrix} \tag{2.77}$$

It can be noted that apart from the additional row and column, the axisymmetric and plane strain stress-strain matrices are identical.

2.13 Three-dimensional stress and strain

When equations (2.50) to (2.52) are extended to the three-dimensional displacement components u, v, and w, three simultaneous partial differential equations equivalent to (2.57) result. Discretisation proceeds as usual, and again the familiar element stiffness properties are derived as

$$[\mathbf{k}_m] = \iiint [\mathbf{B}]^{\mathrm{T}}\,[\mathbf{D}]\,[\mathbf{B}]\;\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \tag{2.78}$$

where the full strain-displacement relations are (Timoshenko and Goodier, 1951),

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{Bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{2.79}$$

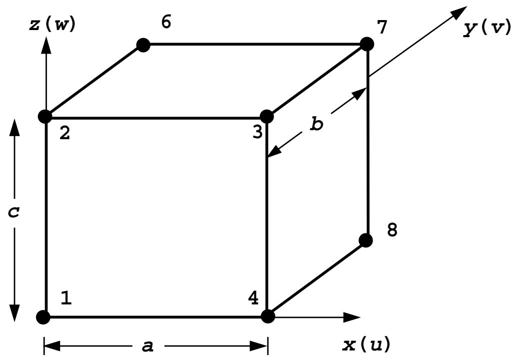


Figure 2.7 8-node "brick" element

or

$$\{\boldsymbol{\epsilon}\} = [\mathbf{A}]\{\mathbf{e}\} \tag{2.80}$$

For example, the 8-noded brick-shaped element shown in Figure 2.7, would have shape functions of the form,

$$[\mathbf{N}] = [N_1\ N_2\ N_3\ N_4\ N_5\ N_6\ N_7\ N_8] \tag{2.81}$$

where

$$\begin{aligned}
N_1 &= \left(1 - \frac{x}{a}\right)\left(1 - \frac{y}{b}\right)\left(1 - \frac{z}{c}\right)\\
N_2 &= \left(1 - \frac{x}{a}\right)\left(1 - \frac{y}{b}\right)\frac{z}{c}\\
N_3 &= \frac{x}{a}\left(1 - \frac{y}{b}\right)\frac{z}{c}\\
N_4 &= \frac{x}{a}\left(1 - \frac{y}{b}\right)\left(1 - \frac{z}{c}\right)\\
N_5 &= \left(1 - \frac{x}{a}\right)\frac{y}{b}\left(1 - \frac{z}{c}\right)\\
N_6 &= \left(1 - \frac{x}{a}\right)\frac{y}{b}\frac{z}{c}\\
N_7 &= \frac{x}{a}\frac{y}{b}\frac{z}{c}\\
N_8 &= \frac{x}{a}\frac{y}{b}\left(1 - \frac{z}{c}\right)
\end{aligned} \tag{2.82}$$

The full [S] matrix would be of the form,

$$[\mathbf{S}] = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & \cdots \\ 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & \cdots \\ 0 & 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & \cdots \end{bmatrix} \tag{2.83}$$

leading as usual to the formation of $[\mathbf{B}] = [\mathbf{A}]\,[\mathbf{S}]$. The elastic stress-strain matrix in three dimensions is given by,

$$[\mathbf{D}] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{2.84}$$

2.14 Plate-bending element

The bending of a thin plate is governed by the equation,

$$D\nabla^4 w = q \tag{2.85}$$

where ∇^4 is the bi-harmonic operator, D is the flexural rigidity of the plate, given by,

$$D = \frac{Eh^3}{12(1-\nu^2)} \tag{2.86}$$

w is the deflection in the transverse z-direction, q is a applied transverse distributed load, and h is the plate thickness.

Solution of (2.85) directly, for example by Galerkin's method, appears to imply that for a fixed D, a thin plate's deflection is unaffected by the value of Poisson's ratio. This is in fact only true for certain boundary conditions and, in general, the integration by parts in the Galerkin process will supply extra terms that are dependent on ν.

This is a case in which the energy approach provides a simpler formulation. The strain energy in a piece of bent plate is given by (Timoshenko and Woinowsky-Krieger, 1959),

$$U = \frac{1}{2}D\iint\left\{\left(\frac{\partial^2 w}{\partial x^2}+\frac{\partial^2 w}{\partial y^2}\right)^2 - 2(1-\nu)\left[\frac{\partial^2 w}{\partial x^2}\frac{\partial^2 w}{\partial y^2}-\left(\frac{\partial^2 w}{\partial x\partial y}\right)^2\right]\right\}\mathrm{d}x\,\mathrm{d}y \tag{2.87}$$

or

$$U = \frac{1}{2}D\iint\left\{\left(\frac{\partial^2 w}{\partial x^2}\right)^2+\left(\frac{\partial^2 w}{\partial y^2}\right)^2 + 2\nu\frac{\partial^2 w}{\partial x^2}\frac{\partial^2 w}{\partial y^2}+2(1-\nu)\left(\frac{\partial^2 w}{\partial x\partial y}\right)^2\right\}\mathrm{d}x\,\mathrm{d}y \tag{2.88}$$

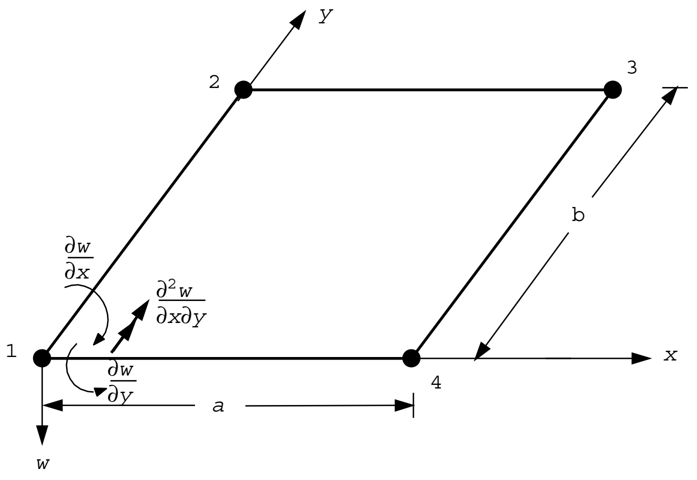

y
2
3
b
$\frac{\partial w}{\partial x}$
$\frac{\partial^2 w}{\partial x \partial y}$
1
$\frac{\partial w}{\partial y}$
4
x
a
w

Figure 2.8 Rectangular plate bending element

Consider, for example, the rectangular element shown in Figure 2.8. If there are assumed to be four degrees of freedom per node, namely

$$w, \quad \theta_x = \frac{\partial w}{\partial x}, \quad \theta_y = \frac{\partial w}{\partial y} \quad \text{and} \quad \theta_{xy} = \frac{\partial^2 w}{\partial x \partial y}$$

then the appropriate element shape functions can be shown to be products of the beam shape functions already described, that is,

$$\tilde{w} = [\mathbf{N}]\{\mathbf{w}\} \tag{2.89}$$

where, if the freedoms $\{\mathbf{w}\}$ are numbered $(w, \theta_x, \theta_y, \theta_{xy})_{\text{node}=1,2,3,4}$, the "first" shape function would be,

$$N_1 = \frac{1}{a^3}(a^3 - 3ax^2 + 2x^3)\frac{1}{b^3}(b^3 - 3by^2 + 2y^3) \tag{2.90}$$

Defining,

$$\begin{aligned}
P_1 &= \frac{1}{a^3}(a^3 - 3ax^2 + 2x^3) \\
Q_1 &= \frac{1}{b^3}(b^3 - 3by^2 + 2y^3) \\
P_2 &= \frac{1}{a^2}(a^2x - 2ax^2 + x^3) \\
Q_2 &= \frac{1}{b^2}(b^2y - 2by^2 + y^3) \\
P_3 &= \frac{1}{a^3}(3ax^2 - 2x^3)
\end{aligned} \tag{2.91}$$

$$Q_3 = \frac{1}{b^3}(3by^2 - 2y^3)$$

$$P_4 = \frac{1}{a^2}(x^3 - ax^2)$$

$$Q_4 = \frac{1}{b^2}(y^3 - by^2)$$

the list of shape functions becomes

$$\begin{aligned} N_1 &= P_1 Q_1 & N_9 &= P_3 Q_3 \\ N_2 &= P_2 Q_1 & N_{10} &= P_4 Q_3 \\ N_3 &= P_1 Q_2 & N_{11} &= P_3 Q_4 \\ N_4 &= P_2 Q_2 & N_{12} &= P_4 Q_4 \\ N_5 &= P_1 Q_3 & N_{13} &= P_3 Q_1 \\ N_6 &= P_2 Q_3 & N_{14} &= P_4 Q_1 \\ N_7 &= P_1 Q_4 & N_{15} &= P_3 Q_2 \\ N_8 &= P_2 Q_4 & N_{16} &= P_4 Q_2 \end{aligned} \tag{2.92}$$

Using the same energy formulation as before, and defining

$$\begin{Bmatrix} M_x \\ M_y \\ M_{xy} \end{Bmatrix} = D \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{Bmatrix} \dfrac{\partial^2}{\partial x^2} \\ \dfrac{\partial^2}{\partial y^2} \\ 2\dfrac{\partial^2}{\partial x \partial y} \end{Bmatrix} w \tag{2.93}$$

or

$$\{\mathbf{M}\} = [\mathbf{D}]\,\{\mathbf{A}\}\,w \tag{2.94}$$

it can readily be verified that (2.88) for strain energy can be written

$$U = \frac{1}{2}\iint (\{\mathbf{A}\}\,w)^{\mathrm{T}}[\mathbf{D}](\{\mathbf{A}\}\,w)\,\mathrm{d}x\,\mathrm{d}y \tag{2.95}$$

and that the stiffness matrix becomes

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}}\,[\mathbf{D}]\,[\mathbf{B}]\,\mathrm{d}x\,\mathrm{d}y \tag{2.96}$$

which is again the familiar equivalent of (2.49) with $[\mathbf{B}] = \{\mathbf{A}\}\,[\mathbf{N}]$.

A typical value of the stiffness matrix is given by

$$k_{m\,i,j} = D\iint \left\{ \frac{\partial^2 N_i}{\partial x^2}\frac{\partial^2 N_j}{\partial x^2} + \nu\frac{\partial^2 N_i}{\partial x^2}\frac{\partial^2 N_j}{\partial y^2} + \nu\frac{\partial^2 N_j}{\partial x^2}\frac{\partial^2 N_i}{\partial y^2} \right.$$
$$\left. + \frac{\partial^2 N_i}{\partial y^2}\frac{\partial^2 N_j}{\partial y^2} + 2(1-\nu)\frac{\partial^2 N_i}{\partial x \partial y}\frac{\partial^2 N_j}{\partial x \partial y} \right\} \mathrm{d}x\,\mathrm{d}y \tag{2.97}$$

and these integrals are performed using Gaussian quadrature in two dimensions in Chapter 4. For some boundary conditions, the terms including Poisson's ratio will cancel out.

2.15 Summary of element equations for solids

The preceding sections have demonstrated the essential similarity of all problems in linear elastic solid mechanics when formulated in terms of finite elements. The statement of element properties is to be found in two expressions, namely, the element stiffness matrix

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}] \, \mathrm{d}x \, \mathrm{d}y \tag{2.98}$$

and the element–mass matrix

$$[\mathbf{m}_m] = \rho \iint [\mathbf{N}]^{\mathrm{T}} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y \tag{2.99}$$

These expressions then appear in the three main classes of problem which concern us in engineering practice, namely

1. Static equilibrium, $$[\mathbf{k}_m] \{\mathbf{u}\} = \{\mathbf{f}\} \tag{2.100}$$

2. Eigenvalue, $$[\mathbf{k}_m] \{\mathbf{a}\} - \omega^2 [\mathbf{m}_m] \{\mathbf{a}\} = \{\mathbf{0}\} \tag{2.101}$$

3. Propagation, $$[\mathbf{k}_m] \{\mathbf{u}\} + [\mathbf{m}_m] \left\{ \frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}t^2} \right\} = \{\mathbf{f}(t)\} \tag{2.102}$$

Static problems lead to simultaneous equations which can be solved for known forces $\{\mathbf{f}\}$ to give equilibrium displacements $\{\mathbf{u}\}$. Eigenvalue problems may be solved by various techniques (iteration, QR algorithm, etc. see Bathe and Wilson, 1996; Jennings and McKeown, 1992) to yield mode shapes $\{\mathbf{a}\}$ and natural frequencies (squared) ω^2 of elastic systems, while propagation problems can be solved by advancing the displacements $\{\mathbf{u}\}$ step by step in time due to a forcing function $\{\mathbf{f}(t)\}$ from known initial conditions.

Later chapters in the book describe programs which enable the user to solve practical engineering problems that are governed by these three basic equations. Additional features, such as treatment of non-linearity, damping etc, will be dealt with in these chapters as they arise.

2.16 Flow of fluids: Navier–Stokes equations

We shall be concerned only with the equations governing the motion of viscous, incompressible fluids. These equations are widely developed elsewhere, for example Schlichting (1960). For two dimensions, preserving an analogy with previous sections on two-dimensional solids, u and v now become velocities in the x and y-directions respectively and ρ is the mass density. Also as before, F_x and F_y are body forces in the appropriate directions.

Conservation of mass leads to

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\rho v) = 0 \tag{2.103}$$

but due to incompressibility this may be reduced to

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{2.104}$$

Conservation of momentum leads to

$$\begin{aligned}
\rho\left(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) &= F_x + \left(\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y}\right) \\
\rho\left(\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) &= F_y + \left(\frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{xy}}{\partial x}\right)
\end{aligned} \tag{2.105}$$

where σ_x, σ_y, and τ_{xy}, are stress components as previously defined for solids.

Introducing the simplest constitutive parameter μ (the molecular viscosity), the following form of the stress equations is reached:

$$\begin{aligned}
\sigma_x &= -p - \frac{2\mu}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + 2\mu\frac{\partial u}{\partial x} \\
\sigma_y &= -p - \frac{2\mu}{3}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + 2\mu\frac{\partial v}{\partial y} \\
\tau_{xy} &= \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)
\end{aligned} \tag{2.106}$$

where p is the fluid pressure.

Combining equations (2.104) to (2.106), a form of the "Navier–Stokes" equations can be written,

$$\begin{aligned}
\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= \frac{1}{\rho}F_x - \frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{1}{3}\frac{\mu}{\rho}\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + \frac{\mu}{\rho}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\
\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= \frac{1}{\rho}F_y - \frac{1}{\rho}\frac{\partial p}{\partial y} + \frac{1}{3}\frac{\mu}{\rho}\frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) + \frac{\mu}{\rho}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)
\end{aligned} \tag{2.107}$$

On introduction of the incompressibility condition, these can be further simplified to

$$\begin{aligned}
\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= \frac{1}{\rho}F_x - \frac{1}{\rho}\frac{\partial p}{\partial x} + \frac{\mu}{\rho}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\
\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} &= \frac{1}{\rho}F_y - \frac{1}{\rho}\frac{\partial p}{\partial y} + \frac{\mu}{\rho}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)
\end{aligned} \tag{2.108}$$

For steady state conditions the terms $\partial u/\partial t$ and $\partial v/\partial t$ can be dropped resulting in "coupled" equations in the "primitive" variables, u, v, and p. The equations are also, in contrast to those of solid elasticity, non-linear because of the presence of products like $u(\partial u/\partial x)$.

Ignoring body forces for the present, the steady state equations to be solved are,

$$u\frac{\partial u}{\partial x}+v\frac{\partial u}{\partial y}+\frac{1}{\rho}\frac{\partial p}{\partial x}-\frac{\mu}{\rho}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)=0$$

$$u\frac{\partial v}{\partial x}+v\frac{\partial v}{\partial y}+\frac{1}{\rho}\frac{\partial p}{\partial y}-\frac{\mu}{\rho}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right)=0 \qquad (2.109)$$

Proceeding as before, and for the moment assuming that the same shape functions are applied to all variables, the following trial solutions, $\tilde{u}=[\mathbf{N}]\{\mathbf{u}\}$, $\tilde{v}=[\mathbf{N}]\{\mathbf{v}\}$ and $\tilde{p}=[\mathbf{N}]\{\mathbf{p}\}$ are used, where $\{\mathbf{u}\}=[u_1\ u_2\ u_3\cdots]^{\mathrm{T}}$, represents the nodal values of the velocity u in the x-direction, etc.

For the purposes of integration during the Galerkin procedure, the terms u and v in (2.109) are set equal to the constants $u=\overline{u}=[\mathbf{N}]\{\mathbf{u}_0\}$ and $v=\overline{v}=[\mathbf{N}]\{\mathbf{v}_0\}$ where $\{\mathbf{u}_0\}$ and $\{\mathbf{v}_0\}$ are estimates of the nodal velocities (see Chapter 9).

After substitution of the trial solutions,

$$\overline{u}\frac{\partial}{\partial x}[\mathbf{N}]\{\mathbf{u}\}+\overline{v}\frac{\partial}{\partial y}[\mathbf{N}]\{\mathbf{u}\}+\frac{1}{\rho}\frac{\partial}{\partial x}[\mathbf{N}]\{\mathbf{p}\}-\frac{\mu}{\rho}\frac{\partial^2}{\partial x^2}[\mathbf{N}]\{\mathbf{u}\}-\frac{\mu}{\rho}\frac{\partial^2}{\partial y^2}[\mathbf{N}]\{\mathbf{u}\}=\{\mathbf{0}\}$$

$$\overline{u}\frac{\partial}{\partial x}[\mathbf{N}]\{\mathbf{v}\}+\overline{v}\frac{\partial}{\partial y}[\mathbf{N}]\{\mathbf{v}\}+\frac{1}{\rho}\frac{\partial}{\partial y}[\mathbf{N}]\{\mathbf{p}\}-\frac{\mu}{\rho}\frac{\partial^2}{\partial x^2}[\mathbf{N}]\{\mathbf{v}\}-\frac{\mu}{\rho}\frac{\partial^2}{\partial y^2}[\mathbf{N}]\{\mathbf{v}\}=\{\mathbf{0}\} \qquad (2.110)$$

Multiplying by the weighting functions and integrating as usual yields

$$\iint[\mathbf{N}]^{\mathrm{T}}\overline{u}\frac{\partial}{\partial x}[\mathbf{N}]\{\mathbf{u}\}\,\mathrm{d}x\,\mathrm{d}x+\iint[\mathbf{N}]^{\mathrm{T}}\overline{v}\frac{\partial}{\partial y}[\mathbf{N}]\{\mathbf{u}\}\,\mathrm{d}x\,\mathrm{d}y+\frac{1}{\rho}\iint[\mathbf{N}]^{\mathrm{T}}\frac{\partial}{\partial x}[\mathbf{N}]\{\mathbf{p}\}\,\mathrm{d}x\,\mathrm{d}y$$
$$-\frac{\mu}{\rho}\iint[\mathbf{N}]^{\mathrm{T}}\frac{\partial^2}{\partial x^2}[\mathbf{N}]\{\mathbf{u}\}\,\mathrm{d}x\,\mathrm{d}y-\frac{\mu}{\rho}\iint[\mathbf{N}]^{\mathrm{T}}\frac{\partial^2}{\partial y^2}[\mathbf{N}]\{\mathbf{u}\}\,\mathrm{d}x\,\mathrm{d}y=\{\mathbf{0}\}$$
$$\iint[\mathbf{N}]^{\mathrm{T}}\overline{u}\frac{\partial}{\partial x}[\mathbf{N}]\{\mathbf{v}\}\,\mathrm{d}x\,\mathrm{d}y+\iint[\mathbf{N}]^{\mathrm{T}}\overline{v}\frac{\partial}{\partial y}[\mathbf{N}]\{\mathbf{v}\}\,\mathrm{d}x\,\mathrm{d}y+\frac{1}{\rho}\iint[\mathbf{N}]^{\mathrm{T}}\frac{\partial}{\partial y}[\mathbf{N}]\{\mathbf{p}\}\,\mathrm{d}x\,\mathrm{d}y$$
$$-\frac{\mu}{\rho}\iint[\mathbf{N}]^{\mathrm{T}}\frac{\partial^2}{\partial x^2}[\mathbf{N}]\{\mathbf{v}\}\,\mathrm{d}x\,\mathrm{d}y-\frac{\mu}{\rho}\iint[\mathbf{N}]^{\mathrm{T}}\frac{\partial^2}{\partial y^2}[\mathbf{N}]\{\mathbf{v}\}\,\mathrm{d}x\,\mathrm{d}y=\{\mathbf{0}\} \qquad (2.111)$$

Integrating products by parts where necessary and neglecting resulting contour integrals gives,

$$\iint[\mathbf{N}]^{\mathrm{T}}\overline{u}\frac{\partial}{\partial x}[\mathbf{N}]\,\mathrm{d}x\,\mathrm{d}x\,\{\mathbf{u}\}+\iint[\mathbf{N}]^{\mathrm{T}}\overline{v}\frac{\partial}{\partial y}[\mathbf{N}]\,\mathrm{d}x\,\mathrm{d}y\,\{\mathbf{u}\}+\frac{1}{\rho}\iint[\mathbf{N}]^{\mathrm{T}}\frac{\partial}{\partial x}[\mathbf{N}]\,\mathrm{d}x\,\mathrm{d}y\,\{\mathbf{p}\}$$
$$+\frac{\mu}{\rho}\iint\frac{\partial}{\partial x}[\mathbf{N}]^{\mathrm{T}}\frac{\partial}{\partial x}[\mathbf{N}]\,\mathrm{d}x\,\mathrm{d}y\,\{\mathbf{u}\}+\frac{\mu}{\rho}\iint\frac{\partial}{\partial y}[\mathbf{N}]^{\mathrm{T}}\frac{\partial}{\partial y}[\mathbf{N}]\,\mathrm{d}x\,\mathrm{d}y\,\{\mathbf{u}\}=\{\mathbf{0}\}$$

$$\iint [\mathbf{N}]^{\mathrm{T}} \bar{u} \frac{\partial}{\partial x} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}x \, \{\mathbf{v}\} + \iint [\mathbf{N}]^{\mathrm{T}} \bar{v} \frac{\partial}{\partial y} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y \, \{\mathbf{v}\} + \frac{1}{\rho} \iint [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial y} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y \, \{\mathbf{p}\}$$
$$+ \frac{\mu}{\rho} \iint \frac{\partial}{\partial x} [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial x} [\mathbf{N}]^{\mathrm{T}} \, \mathrm{d}x \, \mathrm{d}y \, \{\mathbf{v}\} + \frac{\mu}{\rho} \iint \frac{\partial}{\partial y} [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial y} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y \, \{\mathbf{v}\} = \{\mathbf{0}\} \quad (2.112)$$

The set of equations is completed by the continuity condition,

$$\iint [\mathbf{N}]^{\mathrm{T}} \left(\frac{\partial}{\partial x} [\mathbf{N}] \{\mathbf{u}\} + \frac{\partial}{\partial y} [\mathbf{N}] \{\mathbf{v}\} \right) \mathrm{d}x \, \mathrm{d}y = \{\mathbf{0}\} \quad (2.113)$$

Collecting terms in {**u**}, {**p**}, and {**v**} respectively leads to an equilibrium equation (Taylor and Hughes, 1981)

$$\begin{bmatrix} [\mathbf{c}_{11}] & [\mathbf{c}_{12}] & [\mathbf{c}_{13}] \\ [\mathbf{c}_{21}] & [\mathbf{c}_{22}] & [\mathbf{c}_{23}] \\ [\mathbf{c}_{31}] & [\mathbf{c}_{32}] & [\mathbf{c}_{33}] \end{bmatrix} \begin{Bmatrix} \mathbf{u} \\ \mathbf{p} \\ \mathbf{v} \end{Bmatrix} = \begin{Bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{Bmatrix} \quad (2.114)$$

where

$$[\mathbf{c}_{11}] = \iint \left([\mathbf{N}]^{\mathrm{T}} \bar{u} \frac{\partial}{\partial x} [\mathbf{N}] + [\mathbf{N}]^{\mathrm{T}} \bar{v} \frac{\partial}{\partial y} [\mathbf{N}] + \frac{\mu}{\rho} \frac{\partial}{\partial x} [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial x} [\mathbf{N}] + \frac{\mu}{\rho} \frac{\partial}{\partial y} [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial y} [\mathbf{N}] \right) \mathrm{d}x \, \mathrm{d}y$$
$$[\mathbf{c}_{12}] = \frac{1}{\rho} \iint [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial x} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y$$
$$[\mathbf{c}_{13}] = [\mathbf{0}]$$
$$[\mathbf{c}_{21}] = \iint [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial x} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y$$
$$[\mathbf{c}_{22}] = [\mathbf{0}]$$
$$[\mathbf{c}_{23}] = \iint [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial y} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y \quad (2.115)$$
$$[\mathbf{c}_{31}] = [\mathbf{0}]$$
$$[\mathbf{c}_{32}] = \frac{1}{\rho} \iint [\mathbf{N}]^{\mathrm{T}} \frac{\partial}{\partial y} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y$$
$$[\mathbf{c}_{33}] = [\mathbf{c}_{11}]$$

Referring to Table 2.1 we now have many terms of the type $N_i \, \partial N_j / \partial x$ which imply unsymmetrical structures for $[\mathbf{c}_{11}]$ etc., thus special solution algorithms will be necessary. Computational details are left until Chapters 3 and 9. Three-dimensional problems are solved in Chapter 12, in which the velocity component in the z-direction is w.

2.17 Simplified flow equations

In many practical instances it may not be necessary to solve the complete coupled system described in the previous section. The pressure p can be eliminated from (2.108) and if vorticity is defined as,

$$\omega = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} \quad (2.116)$$

this results in a single equation,

$$\frac{\partial \omega}{\partial t} + u\frac{\partial \omega}{\partial x} + v\frac{\partial \omega}{\partial y} = \frac{\mu}{\rho}\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right) \tag{2.117}$$

Defining a stream function ψ such that,

$$\begin{aligned} u &= \frac{\partial \psi}{\partial y} \\ v &= -\frac{\partial \psi}{\partial x} \end{aligned} \tag{2.118}$$

an alternative coupled system involving ψ and ω can be devised, given here for steady state conditions,

$$\begin{aligned} \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} &= \omega \\ \frac{\mu}{\rho}\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right) &= \frac{\partial \psi}{\partial y}\frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x}\frac{\partial \omega}{\partial y} \end{aligned} \tag{2.119}$$

This clearly has the advantage that only two unknowns are involved rather than the previous three. However, the solution of (2.119) is still a relatively complicated process and flow problems are sometimes solved via equation (2.117) alone, assuming that u and v can be approximated by some independent means or measured. In this form, equation (2.117) is an example of the "diffusion–convection" equation, the second order space derivatives corresponding to a "diffusion" process and the first order ones to a "convection" process. The equation arises in various areas of engineering, for example sediment transport and pollutant disposal (Smith, 1976, 1979).

If there is no convection, the resulting equation is of the type

$$\frac{\partial \omega}{\partial t} = \frac{\mu}{\rho}\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right) \tag{2.120}$$

which is the "heat conduction" or "diffusion" equation well known in many areas of engineering.

A final simplification is a reduction to steady state conditions, in which case,

$$\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2} = 0 \tag{2.121}$$

leaving the familiar "Laplace" equation. In the following sections, finite element formulations of these simplified flow equations are described, in order of increasing complexity.

2.17.1 Steady state

The form of Laplace's equation (2.121) which arises in geomechanics, for example concerning 2D groundwater flow beneath a water retaining structure or in an aquifer

(Muskat, 1937) is,

$$k_x \frac{\partial^2 \phi}{\partial x^2} + k_y \frac{\partial^2 \phi}{\partial y^2} = 0 \tag{2.122}$$

where ϕ is the fluid "potential" or total head and k_x and k_y are permeabilities or conductivities in the x and y directions. The finite element discretisation process reduces the differential equation to a set of equilibrium type matrix equations of the form,

$$[\mathbf{k}_c]\{\boldsymbol{\phi}\} = \{\mathbf{q}\} \tag{2.123}$$

where $[\mathbf{k}_c]$ is the symmetrical "conductivity matrix", $\{\boldsymbol{\phi}\}$ is a vector of nodal potential (total head) values, and $\{\mathbf{q}\}$ is a vector of nodal inflows/outflows.

With the usual finite element discretisation,

$$\tilde{\phi} = [\mathbf{N}]\{\boldsymbol{\phi}\} \tag{2.124}$$

reference to Table 2.1 shows that typical terms in the matrix $[\mathbf{k}_c]$ are of the form,

$$\iint \left(k_x \frac{\partial N_i}{\partial x}\frac{\partial N_j}{\partial x} + k_y \frac{\partial N_i}{\partial y}\frac{\partial N_j}{\partial y} \right) \mathrm{d}x\,\mathrm{d}y \tag{2.125}$$

A convenient way of expressing the matrix $[\mathbf{k}_c]$ in (2.123) is,

$$[\mathbf{k}_c] = \iint [\mathbf{T}]^{\mathrm{T}}[\mathbf{K}][\mathbf{T}]\,\mathrm{d}x\,\mathrm{d}y \tag{2.126}$$

where the property matrix $[\mathbf{K}]$ is analogous to the stress-strain matrix $[\mathbf{D}]$ in solid mechanics, where,

$$[\mathbf{K}] = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix} \tag{2.127}$$

(assuming that the principal axes of the permeability tensor coincide with x and y). The $[\mathbf{T}]$ matrix is similar to the $[\mathbf{B}]$ matrix of solid mechanics and is given by (for a 4-node element),

$$[\mathbf{T}] = \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial x} & \frac{\partial N_4}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \frac{\partial N_3}{\partial y} & \frac{\partial N_4}{\partial y} \end{bmatrix} \tag{2.128}$$

The similarity between (2.126) for a fluid and (2.69) for a solid enables the corresponding programs to look similar in spite of the governing differential equations being quite different. This unity of treatment is utilised in describing the programming techniques in Chapter 3.

Finally, it is worth noting that (2.126) can also be arrived at from energy considerations. The equivalent energy statement is that the integral

$$\iint \left[\frac{1}{2}k_x \left(\frac{\partial \phi}{\partial x}\right)^2 + \frac{1}{2}k_y \left(\frac{\partial \phi}{\partial y}\right)^2 \right] \mathrm{d}x\,\mathrm{d}y \tag{2.129}$$

shall be a minimum for all possible $\phi(x, y)$.

Example solutions to steady state problems described by (2.122) are given in Chapter 7. Three-dimensional problems are also solved in Chapter 12.

2.17.2 Transient state

Transient conditions must be analysed in many physical situations, for example in the case of Terzaghi "consolidation" in soil mechanics or transient heat conduction. The governing consolidation diffusion equation for excess pore pressure u_w in 2D, takes the form

$$c_x\frac{\partial^2 u_w}{\partial x^2}+c_y\frac{\partial^2 u_w}{\partial y^2}=\frac{\partial u_w}{\partial t} \tag{2.130}$$

where c_x and c_y are the coefficients of consolidation in the x- and y-directions. Discretisation of the left hand side of (2.130) clearly follows that of (2.122) while the time derivative will be associated with a matrix of the "mass matrix" type from (2.71), without the multiple ρ. Hence, the discretised system is,

$$[\mathbf{k}_c]\{\mathbf{u}_w\}+[\mathbf{m}_m]\left\{\frac{\mathrm{d}\mathbf{u}_w}{\mathrm{d}t}\right\}=\{\mathbf{0}\} \tag{2.131}$$

where $\{\mathbf{u}_w\}$ are the nodal values of u_w.

This set of first order, ordinary differential equations can be solved by many methods, the simplest of which discretise the time derivative by finite differences. The algorithms are described in Chapter 3 with example solutions in Chapters 8 and 12.

2.17.3 Advection

If pollutants, sediments, tracers, etc, are transported by a laminar flow system they are at the same time translated or "advected" by the flow and diffused within it. The governing differential equation for the two-dimensional case is (Smith *et al.*, 1973),

$$c_x\frac{\partial^2\phi}{\partial x^2}+c_y\frac{\partial^2\phi}{\partial y^2}-u\frac{\partial\phi}{\partial x}-v\frac{\partial\phi}{\partial y}=\frac{\partial\phi}{\partial t} \tag{2.132}$$

where ϕ can be interpreted as a "concentration" and u and v are the fluid velocity components in the x- and y-directions (compare equation 2.117).

The extra advection terms $-u\partial\phi/\partial x$ and $-v\partial\phi/\partial y$ compared with (2.130) lead, as shown in Table 2.1, to unsymmetric components of the resulting matrix of the type,

$$\iint\left(-uN_i\frac{\partial N_j}{\partial x}-vN_i\frac{\partial N_j}{\partial y}\right)\mathrm{d}x\,\mathrm{d}y \tag{2.133}$$

which must be added to the symmetric, diffusion components given in (2.125). When this has been done, equilibrium equations like (2.123) or transient equations like (2.131) are regained.

Mathematically, equation (2.132) is a differential equation which is not self-adjoint (Berg, 1962), due to the presence of the first-order spatial derivatives. From a finite element point of view, equations which are not self-adjoint will always lead to unsymmetrical matrices.

A second consequence of non-self-adjoint equations is that there is no energy formulation equivalent to (2.129). It is clearly a benefit of the Galerkin approach that it can be used for all types of equation and is not restricted to self-adjoint systems.

Equation (2.132) can be rendered self-adjoint by using the transformation

$$h = \phi \exp\left(\frac{ux}{2c_x}\right) \exp\left(\frac{vy}{2c_y}\right) \tag{2.134}$$

but this is not recommended unless u and v are small compared with c_x and c_y, as shown by Smith *et al.* (1973).

Equation (2.132) and the use of (2.134) are described in Chapter 8.

2.18 Further coupled equations: Biot consolidation

Thus far in this chapter, analyses of solids and fluids have been considered separately. However, Biot formulated the theory of coupled solid–fluid interaction which finds application in soil mechanics (Smith and Hobbs, 1976). The soil skeleton is treated as a porous elastic solid and the laminar pore fluid is coupled to the solid by the conditions of equilibrium and continuity.

First, for two-dimensional equilibrium in the absence of body forces, the gradient of effective stress from (2.50) must be augmented by the gradients of the fluid pressure u_w as follows:

$$\begin{aligned}
\frac{\partial \sigma_x^{'}}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial u_w}{\partial x} &= 0 \\
\frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y^{'}}{\partial y} + \frac{\partial u_w}{\partial y} &= 0
\end{aligned} \tag{2.135}$$

where $\sigma_x^{'} = \sigma_x - u_w$, etc are "effective" stresses.

Assuming plane strain conditions and small strains, and following the usual sequence of operations for a displacement method, the stress terms in equation (2.135) can be eliminated in terms of displacements to give (Griffiths, 1994),

$$\begin{aligned}
\frac{E^{'}(1-\nu^{'})}{(1+\nu^{'})(1-2\nu^{'})}\left[\frac{\partial^2 u}{\partial x^2} + \frac{(1-2\nu^{'})}{2(1-\nu^{'})}\frac{\partial^2 u}{\partial y^2} + \frac{1}{2(1-\nu^{'})}\frac{\partial^2 v}{\partial x \partial y}\right] + \frac{\partial u_w}{\partial x} &= 0 \\
\frac{E^{'}(1-\nu^{'})}{(1+\nu^{'})(1-2\nu^{'})}\left[\frac{1}{2(1-\nu^{'})}\frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2 v}{\partial y^2} + \frac{(1-2\nu^{'})}{2(1-\nu^{'})}\frac{\partial^2 v}{\partial x^2}\right] + \frac{\partial u_w}{\partial y} &= 0
\end{aligned} \tag{2.136}$$

where $E^{'}$ and $\nu^{'}$ are the effective elastic parameters.

Secondly, from considerations of 2D continuity, and assuming fluid incompressibility, the net flow rate must equal the rate of change of volume of the element of soil, thus

$$\frac{\partial}{\partial t}\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)+\frac{k_x}{\gamma_w}\frac{\partial^2 u_w}{\partial x^2}+\frac{k_y}{\gamma_w}\frac{\partial^2 u_w}{\partial y^2}=0 \tag{2.137}$$

Equations (2.136) and (2.137) represent the coupled “Biot” equations for a 2D poro-elastic material. A solution to these equations will enable the displacements u, v, and excess pore pressure u_w to be estimated at spatial location (x, y) at any time t.

A finite element approach starts by discretising the dependent variables u, v, and excess pore pressure u_w in the normal way, hence

$$\begin{aligned}\tilde{u} &= [\mathbf{N}]\{\mathbf{u}\}\\ \tilde{v} &= [\mathbf{N}]\{\mathbf{v}\}\\ \tilde{u}_w &= [\mathbf{N}]\{\mathbf{u}_w\}\end{aligned} \tag{2.138}$$

In practice, it may be preferable to use a higher order of discretisation for u and v compared with u_w but, for the present, the same shape functions are assumed to describe all three variables.

When discretisation and the Galerkin process are completed, (2.136) and (2.137) lead to the pair of equilibrium and continuity equations:

$$\begin{aligned}[\mathbf{k}_m]\{\mathbf{u}\}+[\mathbf{c}]\{\mathbf{u}_w\} &= \{\mathbf{f}\}\\ [\mathbf{c}]^{\mathrm{T}}\left\{\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right\}-[\mathbf{k}_c]\{\mathbf{u}_w\} &= \{\mathbf{0}\}\end{aligned} \tag{2.139}$$

where, for a 4-noded element,

$$\{\mathbf{u}\}=\begin{Bmatrix}u_1\\v_1\\u_2\\v_2\\u_3\\v_3\\u_4\\v_4\end{Bmatrix} \quad \text{and} \quad \{\mathbf{u}_w\}=\begin{Bmatrix}u_{w_1}\\u_{w_2}\\u_{w_3}\\u_{w_4}\end{Bmatrix} \tag{2.140}$$

$[\mathbf{k}_m]$ and $[\mathbf{k}_c]$ are the familiar elastic stiffness and fluid conductivity matrices respectively, $[\mathbf{c}]$ is a new rectangular coupling matrix consisting of terms of the form

$$\iint \frac{\partial N_j}{\partial x} N_i \,\mathrm{d}x\,\mathrm{d}y \tag{2.141}$$

and $\{\mathbf{f}\}$ is the external loading vector. After assembly into global matrices, equations (2.139) must be integrated in time by some method such as finite differences and this is described further in Chapter 3. Examples of such solutions in practice are given in Chapter 9.

Three-dimensional problems are solved in Chapter 12, in which the displacements in the z-direction are given by w.

2.19 Conclusions

When viewed from a finite element standpoint, all static equilibrium problems, whether involving solids or fluids, take the same form, namely,

$$[\mathbf{k}_m]\{\mathbf{u}\} = \{\mathbf{f}\} \tag{2.142}$$

or

$$[\mathbf{k}_c]\{\boldsymbol{\phi}\} = \{\mathbf{q}\} \tag{2.143}$$

For simple uncoupled problems the solid $[\mathbf{k}_m]$ and fluid $[\mathbf{k}_c]$ matrices have similar symmetrical structures, so computer programs to construct them will also be similar. However, for other problems, for example those described by the Navier–Stokes equations, the constitutive matrices are unsymmetrical and appropriate alternative software will be necessary.

In the same way, eigenvalue, propagation and transient problems all involve the mass matrix $[\mathbf{m}_m]$ (or a simple multiple of it). Therefore, coding of these different types of solutions can be expected to contain sections common to all three problems.

So far, single elements have been considered in the discretisation process, and only the simplest line and rectangular elements have been described. The next chapter is mainly devoted to a description of programming strategy, but before this, the finite element concept is extended to embrace meshes of interlinked elements and elements of general shape.

References

Bathe KJ and Wilson EL 1976 *Numerical Methods in Finite Element Analysis*. Prentice–Hall, Englewood Cliffs, N.J.

Berg PN 1962 *Calculus of Variations*. McGraw–Hill, London, New York.

Biot MA 1941 General theory of three-dimensional consolidation. *J Appl Phys* **12**, 155–164.

Cook RD, Malkus DS and Plesha ME 1989 *Concepts and Applications of Finite Element Analysis*, 3rd edn. John Wiley & Sons, Chichester, New York.

Finlayson BA 1972 *The Method of Weighted Residuals and Variational Principles*. Academic Press, New York.

Griffiths DV 1989 Advantages of consistent over lumped methods for analysis of beams on elastic foundations. *Commun Appl Numer Methods* **5**(1), 53–60.

Griffiths DV 1994 Coupled analyses in geomechanics. In *Visco-Plastic Behavior of Geomaterials* (eds. Cristescu ND and Gioda G). Springer-Verlag, Wien, New York, pp. 245–317. Chapter 5.

Griffiths DV and Smith IM 1991 *Numerical Methods for Engineers*. Blackwell Scientific Publications Ltd., Oxford.

Horne MR and Merchant W 1965 *The Stability of Frames*. Pergamon Press, Oxford.

Jennings A and McKeown JJ 1992 *Matrix Computation*. John Wiley & Sons, Chichester, New York.

Leckie FA and Lindberg GM 1963 The effect of lumped parameters on beam frequencies. *The Aeronaut Q* **14**, 234.

Livesley RK 1975 *Matrix Methods of Structural Analysis*. Pergamon Press, Oxford.

Muskat M 1937 *The Flow of Homogeneous Fluids Through Porous Media*. McGraw-Hill, London, New York.

Rao SS 1989 *The Finite Element Method in Engineering*, 2nd edn. Pergamon Press, Oxford.

Schlichting H 1960 *Boundary Layer Theory*. McGraw-Hill, London, New York.

Smith IM 1976 Integration in time of diffusion and diffusion–convection equations, *Finite Elements in Water Resources*, vol. 1. Pentech Press, Plymouth, Mass., pp. 3–20.

Smith IM 1979 The diffusion–convection equation. *Summary of Numerical Methods for Partial Differential Equations*, Oxford University Press, Oxford, pp. 195-211. Chapter 11.

Smith IM and Hobbs R 1976 Biot analysis of consolidation beneath embankments. *Géotechnique* **26**, 149–171.

Smith IM, Farraday RV and O'Connor BA 1973 Rayleigh-Ritz and Galerkin finite elements for diffusion–convection problems. *Water Resour Res* **9**(3), 593–606.

Strang G and Fix GJ 1973 *An Analysis of the Finite Element Method*. Prentice–Hall, Englewood Cliffs, N.J.

Szabo BA and Lee GC 1969 Derivation of stiffness matrices for problems in plane elasticity by the Galerkin method. *Int J Numer Methods Eng* **1**, 301.

Taig IC 1961 Structural analysis by the matrix displacement method. Technical Report SO17, English Electric Aviation Report, Preston.

Taylor C and Hughes TG 1981 *Finite Elements Programming of the Navier-Stokes Equation*. Pineridge Press, Swansea, UK.

Timoshenko SP and Goodier JN 1982 *Theory of Elasticity*. McGraw-Hill, Singapore. International Edition.

Timoshenko SP and Woinowsky-Krieger S 1959 *Theory of Plates and Shells*. McGraw-Hill, New York.

Zienkiewicz OC and Taylor RL 1989 *The Finite Element Method*, vol. 1, 4th edn. McGraw-Hill, London, New York.

3

Programming Finite Element Computations

3.1 Introduction

In Chapter 2, the finite element spatial discretisation process was described, whereby partial differential equations can be replaced by matrix equations which take the form of linear and non-linear algebraic equations, eigenvalue equations, or ordinary differential equations in the time variable. The present chapter describes how programs can be constructed in order to formulate and solve these kinds of equations.

Before this, two additional features must be introduced. First, we have so far dealt only with the simplest shapes of elements, namely lines and rectangles. Obviously if differential equations are to be solved over regions of general shape, elements must be allowed to assume general shapes as well. This is accomplished by introducing general triangular, quadrilateral, tetrahedral and hexahedral elements together with the concept of a coordinate system local to the element.

Second, we have so far considered only a single element, whereas useful solutions will normally be obtained by many elements, usually from hundreds to millions in practice, joined together at the nodes. Also, various types of boundary conditions may be prescribed which constrain the solution in some way.

Local coordinate systems, multi-element analyses, and incorporation of boundary conditions are all explained in the sections that follow.

3.2 Local coordinates for quadrilateral elements

Figure 3.1 shows two types of plane 4-noded quadrilateral elements. The shape functions for the rectangle (Figure 3.1(a)) were shown to be given by equation (2.60), namely $N_1 = (1 - x/a)(1 - y/b)$ and so on. If it is attempted to construct similar shape functions in the "global" coordinates (x, y) for the general quadrilateral (Figure 3.1(b)), rather complex

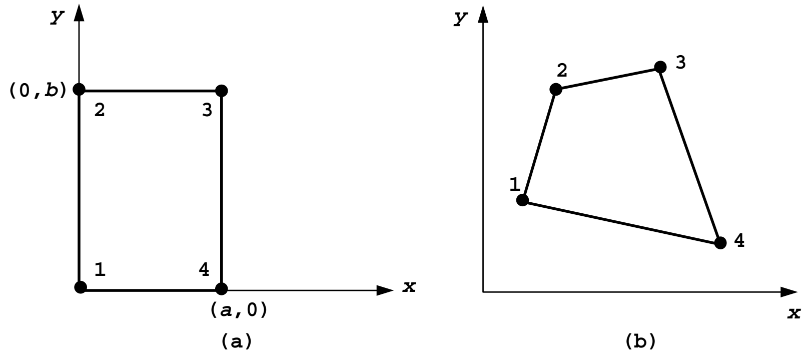


Figure 3.1 (a) Plane rectangular element and (b) Plane general quadrilateral element

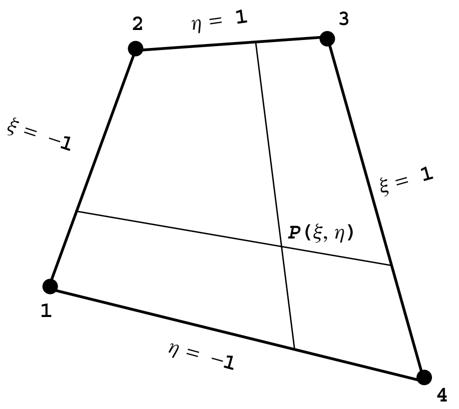


Figure 3.2 Local coordinate system for quadrilateral elements

algebraic expressions will result, which are best generated by computer algebra packages (Griffiths, 1994b, 2004).

Traditionally, the approach has been to work in a local coordinate system as shown in Figure 3.2, originally proposed by Taig (1961), and to evaluate resulting integrals numerically. The general point $P(\xi, \eta)$ within the quadrilateral is located at the intersection of two lines which cut opposite sides of the quadrilateral in equal proportions. For reasons associated with subsequent numerical integrations it proves to be convenient to "normalise" the coordinates so that side 12 has $\xi = -1$, side 34 has $\xi = 1$, side 41 has $\eta = -1$, and side 23 has $\eta = 1$. In this system, the intersection of the bisectors of opposite sides of the quadrilateral is the point $(0, 0)$, while the corners 1, 2, 3, and 4 are $(-1, -1)$, $(-1, 1)$, $(1, 1)$, and $(1, -1)$ respectively.

When this choice is adopted, the shape functions for a 4-noded quadrilateral with corner nodes take the simple form

$$N_1 = \frac{1}{4}(1 - \xi)(1 - \eta)$$

$$
\begin{aligned}
N_2 &= \frac{1}{4}(1-\xi)(1+\eta) \\
N_3 &= \frac{1}{4}(1+\xi)(1+\eta) \\
N_4 &= \frac{1}{4}(1+\xi)(1-\eta)
\end{aligned}
\tag{3.1}
$$

and these can be used to describe the variation of unknowns such as displacement or fluid potential in an element as before.

The same shape functions can also often be used to specify the relation between the global (x, y) and local (ξ, η) coordinate systems. If this is so the element is of a type called "isoparametric" (Ergatoudis *et al.*, 1968; Zienkiewicz *et al.*, 1969), and the 4-node quadrilateral is an example. The coordinate transformation is therefore,

$$
\begin{aligned}
x &= N_1x_1 + N_2x_2 + N_3x_3 + N_4x_4 \\
&= [\mathbf{N}]\{\mathbf{x}\} \\
y &= N_1y_1 + N_2y_2 + N_3y_3 + N_4y_4 \\
&= [\mathbf{N}]\{\mathbf{y}\}
\end{aligned}
\tag{3.2}
$$

where the $[\mathbf{N}]$ are given by (3.1) and $\{\mathbf{x}\}$ and $\{\mathbf{y}\}$ are the nodal coordinates.

In the previous chapter (e.g. equations 2.68 and 2.128), it was shown that element properties involve not only $[\mathbf{N}]$ but also their derivatives with respect to the global coordinates (x, y) which appear in matrices such as $[\mathbf{B}]$ and $[\mathbf{T}]$. Further, products of these quantities need to be integrated over the element area or volume.

Derivatives are easily converted from one coordinate system to the other by means of the chain rule of partial differentiation, best expressed in matrix form for two dimensions by

$$
\left\{\begin{matrix} \dfrac{\partial}{\partial\xi} \\ \dfrac{\partial}{\partial\eta} \end{matrix}\right\} = \begin{bmatrix} \dfrac{\partial x}{\partial\xi} & \dfrac{\partial y}{\partial\xi} \\ \dfrac{\partial x}{\partial\eta} & \dfrac{\partial y}{\partial\eta} \end{bmatrix} \left\{\begin{matrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{matrix}\right\} = [\mathbf{J}] \left\{\begin{matrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{matrix}\right\}
\tag{3.3}
$$

or

$$
\left\{\begin{matrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \end{matrix}\right\} = [\mathbf{J}]^{-1} \left\{\begin{matrix} \dfrac{\partial}{\partial\xi} \\ \dfrac{\partial}{\partial\eta} \end{matrix}\right\}
\tag{3.4}
$$

where $[\mathbf{J}]$ is the Jacobian matrix. The determinant of this matrix, $\det|\mathbf{J}|$ known as "The Jacobian", must also be evaluated because it is used in the transformed integrals as follows:

$$
\iint \mathrm{d}x\,\mathrm{d}y = \int_{-1}^{1}\int_{-1}^{1} \det|\mathbf{J}|\;\mathrm{d}\xi\,\mathrm{d}\eta
\tag{3.5}
$$

For three dimensions, the equivalent expressions are self-evident.

Degenerate quadrilaterals such as the one shown in Figure 3.3(a) are usually acceptable, however reflex interior angles as shown in Figure 3.3(b) should be avoided as this will cause the Jacobian to become indeterminate.

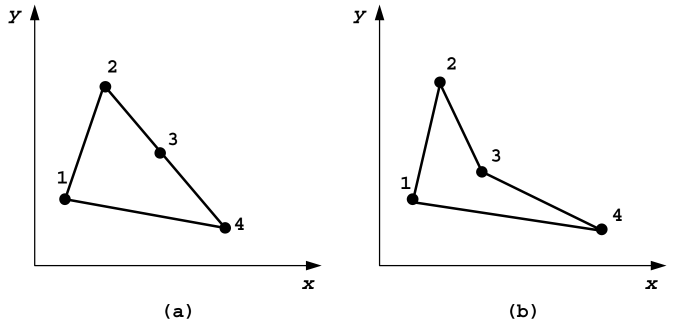


Figure 3.3 (a) Degenerate quadrilateral (b) Unacceptable quadrilateral

3.2.1 Numerical integration for quadrilaterals

Although some integrals of this type can be evaluated analytically, this has traditionally been impractical for complicated functions, particularly in the general case when (ξ, η) become curvilinear (Ergatoudis *et al.*, 1968). In most finite element programs (3.5) are evaluated numerically, using Gauss–Legendre quadrature over quadrilateral regions (Irons, 1966a,b). The quadrature rules in two dimensions are all of the form

$$\begin{aligned}\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)\det|\mathbf{J}|\,\mathrm{d}\xi\,\mathrm{d}\eta &\approx \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j f(\xi_i,\eta_j)\\ &\approx \sum_{i=1}^{\mathtt{nip}} W_i f(\xi,\eta)_i \qquad (3.6)\end{aligned}$$

where $\mathtt{nip} = n^2$ (total number of integrating points), w_i and w_j (or $W_i = w_i w_j$) are weighting coefficients and (ξ_i, η_j) are sampling points within the element. These values for n equal to 1, 2, and 3 are shown in Table 3.1, and complete tables are available in other sources, for example Kopal (1961). The table assumes that the range of integration is ± 1, hence the reason for normalising the local coordinate system in this way.

The approximate equality in (3.6) is exact for cubic functions when $n = 2$ and for quintics when $n = 3$. Usually one attempts to perform integrations over finite elements as accurately as possible, but in special circumstances (Zienkiewicz *et al.*, 1971) "reduced" integration, whereby integrals are deliberately evaluated approximately by decreasing n can improve the quality of solutions.

3.2.2 Analytical integration for quadrilaterals

Computer Algebra Systems (CAS) such as "REDUCE" and "Maple" enable algebraic expressions (e.g. the finite element shape functions) to be manipulated essentially "analytically". Expressions can be differentiated, integrated, factorised and so on, leading to explicit formulations of element matrices avoiding the need for conventional numerical integration. Particularly for three-dimensional elements, this approach can lead to substantial savings

Table 3.1 Coordinates and weights in Gauss–Legendre quadrilateral integration formulae

n	nip	(ξ_i, η_j)	w_i, w_j	W_i
1	1	$(0, 0)$	$(2, 2)$	4
2	4	$\left(\pm\sqrt{\frac{1}{3}}, \pm\sqrt{\frac{1}{3}}\right)$	$(1, 1)$	1
3	9(4@)	$\left(\pm\sqrt{\frac{3}{5}}, \pm\sqrt{\frac{3}{5}}\right)$	$\left(\frac{5}{9}, \frac{5}{9}\right)$	$\frac{25}{81}$
	(2@)	$\left(\pm\sqrt{\frac{3}{5}}, 0\right)$	$\left(\frac{5}{9}, \frac{8}{9}\right)$	$\frac{40}{81}$
	(2@)	$\left(0, \pm\sqrt{\frac{3}{5}}\right)$	$\left(\frac{8}{9}, \frac{5}{9}\right)$	$\frac{40}{81}$
	(1@)	$(0, 0)$	$\left(\frac{8}{9}, \frac{8}{9}\right)$	$\frac{64}{81}$

in integration times. A further point is that for some elements (e.g. a 14-node hexahedron described later in this Chapter) the shape functions are so complex algebraically that it is doubtful if they could be isolated at all without the help of computer algebra.

For finite elements in the context of plane elasticity, the element stiffness matrix has been shown in Chapter 2 (e.g. 2.69) to be given by integrals of the form

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}] \, \mathrm{d}x \, \mathrm{d}y \tag{3.7}$$

where [**B**] and [**D**] represent the strain-displacement and stress–strain matrices respectively.

In the case of quadrilateral elements, if the element is rectangular with its sides parallel to the x- and y-axes, the term under the integral consists of simple polynomial terms which can be easily integrated in closed form by separation of the variables, resulting in compact terms like (2.63). In general however, quadrilateral elements will lead to very complicated expressions under the integral sign which can only be tackled numerically.

Noting that “2-point” Gaussian quadrature, that is `nip = 4`, leads in most cases to accurate estimates of the stiffness matrix of a 4-node general quadrilateral, a compromise approach is to evaluate the contribution to the stiffness matrix coming from each of the four “Gauss-points” algebraically and add them together, thus:

$$[\mathbf{k}_m] \approx \sum_{i=1}^{4} W_i \det|\mathbf{J}|_i \left([\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}]\right)_i \tag{3.8}$$

where $\det|\mathbf{J}|$ is the Jacobian described previously.

This at first leads to rather long expressions, but a considerable amount of cancelling and simplification is possible (e.g. the $1/\sqrt{3}$ term that appears in the sampling points of the integration formula disappears in the simplification process). The algebraic expressions can be generated with the help of a CAS and the risk of typographical errors can be virtually eliminated by outputting the results in Fortran format.

The simplified algebraic expressions that form the stiffness matrix of the 4-node quadrilateral element by this method have been isolated, and form the basis of subroutine `stiff4` used in Program 11.5 of this book. A detailed description of the method is given in Griffiths (1994b).

The same technique can be applied to other element types (Cardoso 1994) and other element matrices (e.g. 8-node quadrilaterals, 3D elements, mass, conductivity, etc). For example, the technique is to be found again in Program 7.3, where the conductivity matrix of a general 4-node quadrilateral element is computed algebraically using subroutine `seep4`.

A similar approach was used to create subroutine `bee8` used in Programs 6.3, 6.8 and 6.9, which generates an algebraic version of the [**B**] matrix for a general 8-node quadrilateral element, corresponding to any given local coordinate (ξ, η).

3.3 Local coordinates for triangular elements

Local coordinates for triangles are conveniently described in terms of a right-angled isosceles triangle of side length equal to unity as shown in Figure 3.4. This approach is exactly equivalent to "area coordinates" (Zienkiewicz *et al.*, 1971) in which any point within the triangle can be referenced using local coordinates (L_1, L_2). Clearly for a plane region, only two independent coordinates are necessary. However a third "coordinate" L_3 given by,

$$L_3 = 1 - L_1 - L_2 \tag{3.9}$$

can sometimes be included to simplify the algebra.

For example, the shape functions for a 3-noded ("constant strain") triangular element (Figure 3.4(b)) take the form

$$\begin{aligned} N_1 &= L_1 \\ N_2 &= L_3 \\ N_3 &= L_2 \end{aligned} \tag{3.10}$$

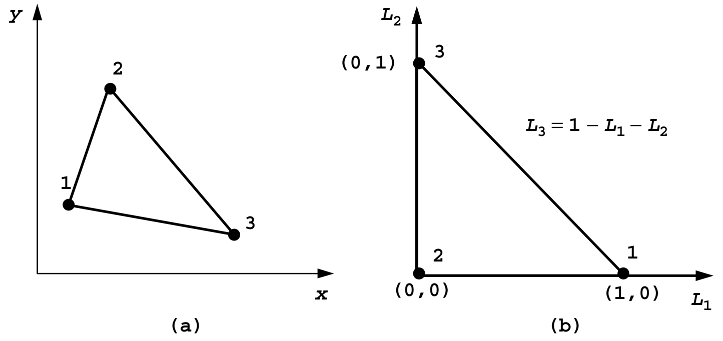


Figure 3.4 (a) General triangular element (b) Local coordinate system for triangular elements

and as before the isoparametric property gives,

$$\begin{aligned} x &= N_1x_1 + N_2x_2 + N_3x_3 \\ &= [\mathbf{N}]\{\mathbf{x}\} \\ y &= N_1y_1 + N_2y_2 + N_3y_3 \\ &= [\mathbf{N}]\{\mathbf{y}\} \end{aligned} \tag{3.11}$$

Equations (3.3) and (3.4) from the previous paragraph still apply regarding the Jacobian matrix but equation (3.5) must be modified for triangles to give,

$$\iint \mathrm{d}x\,\mathrm{d}y = \int_0^1 \int_0^{1-L_1} \det|\mathbf{J}|\;\mathrm{d}L_2\,\mathrm{d}L_1 \tag{3.12}$$

3.3.1 Numerical integration for triangles

Numerical integration over triangular regions is similar to that for quadrilaterals, and takes the general form

$$\int_0^1 \int_0^{1-L_1} f(L_1, L_2)\,\mathrm{d}L_2\,\mathrm{d}L_1 \approx \sum_{i=1}^{\mathtt{nip}} W_i f(L_1, L_2)_i \tag{3.13}$$

where W_i is the weighting coefficient corresponding to the sampling point $(L_1, L_2)_i$ and `nip` represents the number of integrating points. Typical values of the weights and sampling points are given in Table 3.2.

As with quadrilaterals, numerical integration can be exact for certain polynomials. For example, in Table 3.2, the 1-point rule is exact for integration of first degree polynomials and the 3-point rule is exact for polynomials of second degree. Reduced integration can again be beneficial in some instances.

Computer formulations involving local coordinates, transformations of coordinates and numerical integration are described in subsequent paragraphs.

Table 3.2 Coordinates and weights in triangular integration formulae

`nip`	$(L_1, L_2)_i$	W_i
1	$\left(\frac{1}{3}, \frac{1}{3}\right)$	$\frac{1}{2}$
3	$\left(\frac{1}{2}, \frac{1}{2}\right)$	$\frac{1}{6}$
	$\left(\frac{1}{2}, 0\right)$	$\frac{1}{6}$
	$\left(0, \frac{1}{2}\right)$	$\frac{1}{6}$

3.4 Multi-element assemblies

Properties of elements in isolation have been shown to be given by matrix equations, for example the conductivity equation (2.123),

$$[\mathbf{k}_c]\{\boldsymbol{\phi}\} = \{\mathbf{q}\} \tag{3.14}$$

describing steady laminar fluid flow. Figure 3.5 shows a small mesh containing three quadrilateral elements, all of which have properties defined by (3.14). If an assembly strategy is chosen (for non-assembly strategies see Section 3.5), the next task is to assemble the elements and so derive the properties of the 3-element "global" system. Each element possesses node numbers, not circled, which follow the scheme in Figure 3.1(b), namely numbering clockwise starting at any corner. Since there is only one unknown at every node, the fluid "potential", each individual element equation can be written (omitting the c-subscript for clarity),

$$\begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & k_{1,4} \\ k_{2,1} & k_{2,2} & k_{2,3} & k_{2,4} \\ k_{3,1} & k_{3,2} & k_{3,3} & k_{3,4} \\ k_{4,1} & k_{4,2} & k_{4,3} & k_{4,4} \end{bmatrix} \begin{Bmatrix} \phi_1 \\ \phi_2 \\ \phi_3 \\ \phi_4 \end{Bmatrix} = \begin{Bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{Bmatrix} \tag{3.15}$$

However, in the mesh numbering system circled, mesh node 4 corresponds to element node 1 of element 1 and to element node 2 of element 3. The total number of equations for the mesh is 8 and, within this system, term $k_{1,1}$ from element 1 and term $k_{2,2}$ from element 3 would be added together to give global term $K_{4,4}$ and so on. The global matrix for Figure 3.5 is given in Table 3.3, where the superscripts refer to element numbers.

The global matrix equation can be written as

$$[\mathbf{K}_c]\{\boldsymbol{\Phi}\} = \{\mathbf{Q}\} \tag{3.16}$$

where the upper case notation emphasises that these are global (assembled) equations.

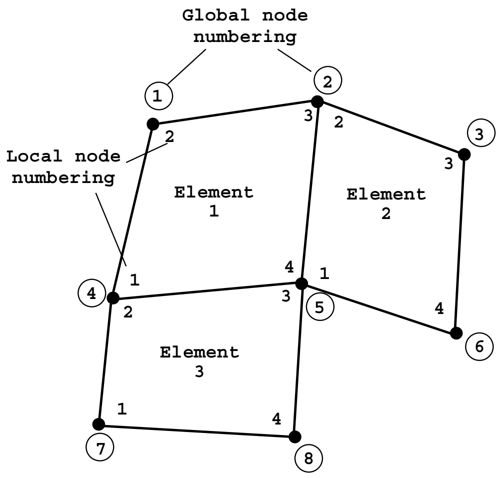


Figure 3.5 Mesh of quadrilateral elements

Table 3.3 Global matrix assembly for mesh in Figure 3.5. Superscripts indicate element numbers

$k^1_{2,2}$	$k^1_{2,3}$	0	$k^1_{2,1}$	$k^1_{2,4}$	0	0	0
$k^1_{3,2}$	$k^1_{3,3}+k^2_{2,2}$	$k^2_{2,3}$	$k^1_{3,1}$	$k^1_{3,4}+k^2_{2,1}$	$k^2_{2,4}$	0	0
0	$k^2_{3,2}$	$k^2_{3,3}$	0	$k^2_{3,1}$	$k^2_{3,4}$	0	0
$k^1_{1,2}$	$k^1_{1,3}$	0	$k^1_{1,1}+k^3_{2,2}$	$k^1_{1,4}+k^3_{2,3}$	0	$k^3_{2,1}$	$k^3_{2,4}$
$k^1_{4,2}$	$k^1_{4,3}+k^2_{1,2}$	$k^2_{1,3}$	$k^1_{4,1}+k^3_{3,2}$	$k^1_{4,4}+k^2_{1,1}+k^3_{3,3}$	$k^2_{1,4}$	$k^3_{3,1}$	$k^3_{3,4}$
0	$k^2_{4,2}$	$k^2_{4,3}$	0	$k^2_{4,1}$	$k^2_{4,4}$	0	0
0	0	0	$k^3_{1,2}$	$k^3_{1,3}$	0	$k^3_{1,1}$	$k^3_{1,4}$
0	0	0	$k^3_{4,2}$	$k^3_{4,3}$	0	$k^3_{4,1}$	$k^3_{4,4}$

This system or global matrix is symmetrical provided its constituent matrices are symmetrical. The matrix also possesses the useful property of "bandedness", which means that the non-zero terms are concentrated around the "leading diagonal" which stretches from the upper left to the lower right of the table. In this example, no term in any row can be more than four locations removed from the leading diagonal, so the system is said to have a "semi-bandwidth" of `nband = 4`. This can be obtained by inspection from Figure 3.5 by subtracting the lowest from the highest global freedom number in each element.

The importance of efficient mesh numbering is illustrated for a mesh of line elements in Figure 3.6 where the scheme in parentheses has `nband = 13` compared to the scheme using circles with `nband = 2`.

If system symmetry exists it should also be taken into account. Using a constant bandwidth storage strategy, the system in Table 3.3 would require 40 storage locations (eight equations times five terms on each line). Greater efficiency can be achieved through "skyline" storage (Bathe, 1996), where the variability of the bandwidth is taken into account, requiring 27 storage locations in this case. Most of the programs described in this book make use of this variable bandwidth or "skyline" storage strategy (see Figure 3.18 for examples of different storage strategies).

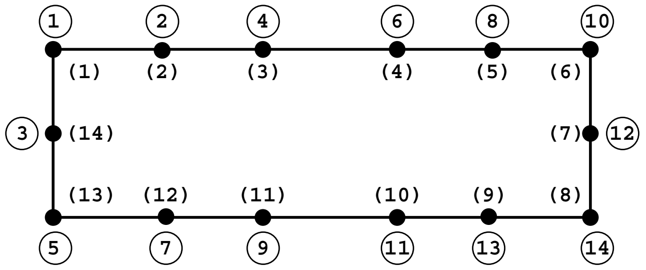


Figure 3.6 Alternative mesh numbering schemes

Later in this chapter, subroutines are described, whereby global matrices like that in Table 3.3 can be automatically assembled in band or "skyline" form, from the constituent element matrices.

3.5 "Element-by-element" or "Mesh-free" techniques

Our purpose is to solve classes of problems, for example as summarised for solids by equations (2.100) to (2.102) for a single element by

1. Static equilibrium problems, $$[\mathbf{k}_m]\{\mathbf{u}\} = \{\mathbf{f}\} \tag{3.17}$$
2. Eigenvalue problems, $$[\mathbf{k}_m]\{\mathbf{a}\} - \omega^2[\mathbf{m}_m]\{\mathbf{a}\} = \{\mathbf{0}\} \tag{3.18}$$
3. Propagation problems, $$[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{m}_m]\left\{\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}t^2}\right\} = \{\mathbf{f}(t)\} \tag{3.19}$$

Traditionally, computer programs have been based on the assembly techniques described in the previous section. For static equilibrium problems, all the element $[\mathbf{k}_m]$ matrices and $\{\mathbf{f}\}$ vectors would be assembled to form a "global" system of linear simultaneous equations of the form

$$[\mathbf{K}_m]\{\mathbf{U}\} = \{\mathbf{F}\} \tag{3.20}$$

Then the assembled global linear algebraic system would be solved, typically by some form of Gaussian elimination. In the previous section, it was emphasised that this strategy depends on efficient storage of the system coefficient matrices. In Gaussian elimination processes, "fill-in" means that coefficients in Table 3.3 like the fourth one in the third row, will not remain zero during the elimination process. Therefore, all coefficients contained within a "band" or "skyline" must be stored and manipulated.

As problem sizes grow, this storage requirement becomes a burden, even on a modern computer. For meshes of three-dimensional elements 100,000 equations are likely to have a semi-bandwidth of the order of 1000. Thus 10^8 "words" of storage, typically 800 Mb would be required to hold the coefficient matrix $[\mathbf{K}_m]$. If this space is not available, out-of-memory techniques or "paging" (see Chapter 1) cause a serious deterioration in analysis speeds.

For this reason, alternative solution strategies to Gaussian elimination have been sought, and there has been a resurgence of interest in iterative techniques for the solution of large systems like (3.20). Griffiths and Smith (1991) describe a number of algorithms of this type, the most popular for symmetric positive definite systems being based on the method of "conjugate gradients" (Jennings and McKeown, 1992).

3.5.1 Conjugate gradient method

Solution of the linear algebraic system (3.20) starts by setting

$$\{\mathbf{P}\}_0 = \{\mathbf{R}\}_0 = \{\mathbf{F}\} - [\mathbf{K}_m]\{\mathbf{U}\}_0 \tag{3.21}$$

where $\{\mathbf{R}\}_0$ is the "residual" or error for a first trial $\{\mathbf{U}\}_0$, followed by k steps of the process:

$$\begin{aligned}
\{\mathbf{Q}\}_k &= [\mathbf{K}_m]\,\{\mathbf{P}\}_k \\
\alpha_k &= \frac{\{\mathbf{R}\}_k^{\mathrm{T}}\,\{\mathbf{R}\}_k}{\{\mathbf{P}\}_k^{\mathrm{T}}\,\{\mathbf{Q}\}_k} \\
\{\mathbf{U}\}_{k+1} &= \{\mathbf{U}\}_k + \alpha_k\,\{\mathbf{P}\}_k \\
\{\mathbf{R}\}_{k+1} &= \{\mathbf{R}\}_k - \alpha_k\,\{\mathbf{U}\}_k \\
\beta_k &= \frac{\{\mathbf{R}\}_{k+1}^{\mathrm{T}}\,\{\mathbf{R}\}_{k+1}}{\{\mathbf{R}\}_k^{\mathrm{T}}\,\{\mathbf{R}\}_k} \\
\{\mathbf{P}\}_{k+1} &= \{\mathbf{R}\}_{k+1} + \beta_k\,\{\mathbf{R}\}_k
\end{aligned} \tag{3.22}$$

until the difference between $\{\mathbf{U}\}_{k+1}$ and $\{\mathbf{U}\}_k$ is sufficiently small, as determined by a convergence criterion. In the above $\{\mathbf{Q}\}$, $\{\mathbf{P}\}$, and $\{\mathbf{R}\}$ are vectors of length equal to the number of equations to be solved (`neq` in programming terminology), while α and β are scalars.

It can be seen that the algorithm described by equations (3.21) and (3.22) consists of simple vector operations of the type $\{\mathbf{U}\} + \alpha\,\{\mathbf{P}\}$, which are neatly coded in Fortran 95 using whole arrays (Chapter 1), inner products of the type $\{\mathbf{R}\}^{\mathrm{T}}\,\{\mathbf{R}\}$ which are computed by the Fortran 95 intrinsic procedure `DOT_PRODUCT`, and a single two-dimensional array operation $\{\mathbf{Q}\} = [\mathbf{K}_m]\,\{\mathbf{P}\}$ which can be computed by the Fortran 95 intrinsic procedure `MATMUL`.

Vitally, however, if $[\mathbf{K}_m]$ is a system stiffness matrix such as in (3.20) or Table 3.3, and all that is required is the product $[\mathbf{K}_m]\,\{\mathbf{P}\}$ where $\{\mathbf{P}\}$ is a known vector, this product can be carried out "element-by-element" without ever assembling $[\mathbf{K}_m]$ at all. That is,

$$\{\mathbf{Q}\} = \sum_{i=1}^{\mathtt{nels}} [\mathbf{k}_m]_i\,\{\mathbf{p}\}_i \tag{3.23}$$

where `nels` is the number of elements, $[\mathbf{k}_m]_i$ is the element stiffness matrix of the ith element and $\{\mathbf{p}\}_i$ the appropriate part of $\{\mathbf{P}\}$, gathered as $\begin{bmatrix} p_7 & p_4 & p_5 & p_8 \end{bmatrix}^{\mathrm{T}}$ for $i = 3$ in Figure 3.5 and so on.

The storage required by such an algorithm, compared with the 800 Mb discussed earlier for a 3D system of 100,000 unknowns would be an order of magnitude less, and would grow linearly with the increase in number of elements or equations rather than as the square. In practice "preconditioning" (Griffiths and Smith, 1991) can be used to accelerate convergence of some iterative processes for solving the "element-by-element" version of (3.20). Iterative strategies for the solution of equations of the type given by (3.18) and (3.19) will also be described in due course.

3.5.2 Preconditioning

Iterative solution of (3.20) can be accelerated by use of a "preconditioner" matrix $[\mathbf{P}]$, such that

$$[\mathbf{P}][\mathbf{K}_m]\,\{\mathbf{U}\} = [\mathbf{P}]\,\{\mathbf{F}\} \tag{3.24}$$

or

$$[\mathbf{K}_m][\mathbf{P}]\{\mathbf{U}\} = [\mathbf{P}]\{\mathbf{F}\} \tag{3.25}$$

With excessive computational effort, [**P**] could be calculated as the inverse of $[\mathbf{K}_m]$ and the solution obtained in one step as,

$$\{\mathbf{U}\} = [\mathbf{K}_m]^{-1}\{\mathbf{F}\} \tag{3.26}$$

In practice it turns out that relatively crude approximations to $[\mathbf{K}_m]^{-1}$ can be used to construct [**P**] and hence be used in the iteration process. For example, "diagonal" preconditioning uses the inverse of the diagonal terms of $[\mathbf{K}_m]$ as a vector {**P**}. This approach is easy to program and carries over easily to the parallel solutions in Chapter 12. Alternatively, element-by-element preconditioning (Hughes *et al.*, 1983; Smith *et al.*, 1989) can be exploited, which also carries over in parallel.

3.5.3 Unsymmetric systems

It was shown in Section 2.16 that finite element discretisation of the Navier–Stokes equations leads to unsymmetric element matrices which, if assembled, result in unsymmetric global systems of equations. When these are solved (see Chapter 9) appropriate Gaussian elimination solvers have to be used.

In an element-by-element context, we therefore seek equivalent iterative techniques to the conjugate gradient processes described above for symmetric systems. The essential feature that such a technique must possess is that it consists only of matrix–vector multiplications which can be carried out by (3.23) together with vector operations and inner products which are readily parallelisable.

Kelley (1995) and Greenbaum (1997) have described variations on this theme. Typical methods are:

GMRES	Generalised minimum residual
BiCGStab	Stabilised bi-conjugate gradient
BiCGStab(l)	Stabilised hybrid bi-conjugate gradient

all of which have been applied to finite element systems by Smith (2000), who includes code for both left- and right-preconditioned BiCGStab following (3.24) and (3.25).

In this book, the method selected is BiCGStab(l), described by Sleijpen *et al.* (1994) for example. The BiCGStab algorithm is

$$\{\mathbf{P}\}_0 = \{\mathbf{R}\}_0 = \{\mathbf{F}\} - [\mathbf{K}_m]\{\mathbf{U}\}_0 \tag{3.27}$$

where $\{\mathbf{R}\}_0$ is the "residual" or error for a first trial $\{\mathbf{U}\}_0$.

We then choose a vector $\{\hat{\mathbf{R}}_0\}$ such that $\{\mathbf{R}\}_0^{\mathrm{T}}\{\hat{\mathbf{R}}_0\} \neq 0$, followed by k steps of the process:

$$\text{(a)} \quad \{\mathbf{Q}\}_{k-1} = [\mathbf{K}_m]\{\mathbf{P}\}_{k-1}$$

$$\{\mathbf{U}\}_{k-\frac{1}{2}} = \{\mathbf{U}\}_{k-1} + \alpha_{k-1}\{\mathbf{P}\}_{k-1}$$

$$\text{where} \quad \alpha_{k-1} = \frac{\{\mathbf{R}\}^{\mathrm{T}}_{k-1}\{\hat{\mathbf{R}}_0\}}{\{\mathbf{Q}\}^{\mathrm{T}}_{k-1}\{\hat{\mathbf{R}}_0\}} \tag{3.28}$$

$$\{\mathbf{R}\}_{k-\frac{1}{2}} = \{\mathbf{R}\}_{k-1} - \alpha_{k-1}\{\mathbf{Q}\}_{k-1}$$

$$\begin{aligned} \text{(b)} \quad \{\mathbf{S}\}_{k-\frac{1}{2}} &= [\mathbf{K}_m]\{\mathbf{R}\}_{k-\frac{1}{2}} \\ \{\mathbf{U}\}_k &= \{\mathbf{U}\}_{k-\frac{1}{2}} + \omega_k\{\mathbf{R}\}_{k-\frac{1}{2}} \\ \text{where} \quad \omega_k &= \frac{\{\mathbf{R}\}^{\mathrm{T}}_{k-\frac{1}{2}}\{\mathbf{S}\}_{k-\frac{1}{2}}}{\{\mathbf{S}\}^{\mathrm{T}}_{k-\frac{1}{2}}\{\mathbf{S}\}_{k-\frac{1}{2}}} \\ \{\mathbf{R}\}_k &= \{\mathbf{R}\}_{k-\frac{1}{2}} - \omega_k\{\mathbf{S}\}_{k-\frac{1}{2}} \\ \beta_k &= \frac{\alpha_{k-1}\{\mathbf{R}\}^{\mathrm{T}}_k\{\hat{\mathbf{R}}_0\}}{\omega_k\{\mathbf{R}\}^{\mathrm{T}}_{k-1}\{\hat{\mathbf{R}}_0\}} \\ \{\mathbf{P}\}_k &= \{\mathbf{R}\}_k + \beta_k\{\mathbf{P}\}_{k-1} - \beta_k\omega_k\{\mathbf{Q}\}_{k-1} \end{aligned} \tag{3.29}$$

until convergence is achieved. Compared with (3.21) and (3.22), we see a similar, but two stage process with initialisation followed by stages (a) and (b) in both of which a matrix–vector multiplication like (3.23) is involved, together with whole array operations and inner products, readily computed in Fortran 95.

The hybrid BiCGStab(l) version (Sleijpen *et al.*, 1994) involves essentially the same arithmetic. Serial implementations involving it can be found in Chapter 9 and parallel equivalents in Chapter 12.

3.5.4 Symmetric non-positive definite equations

When Biot's equations for coupled consolidation (2.136, 2.137) are discretised by finite elements as (3.112) or (3.115) these systems will be seen to be symmetric but non-positive definite. Although a candidate solution algorithm is the minimum residual method (MINRES), Smith (2000) found that the diagonally preconditioned conjugate gradient method worked quite effectively and is used herein. However, the whole question of preconditioning is a developing area (Chan *et al.*, 2001).

3.5.5 Symmetric eigenvalue systems

Again in an element-by-element context we seek algorithms which have at their heart matrix–vector operations like (3.23) and vector or inner product operations which can readily be parallelised. Candidates (Bai *et al.*, 2000) are the long-established Lanczos method and the Jacobi-Davidson method in which interest has recently been revived (Sleijpen and van der Vorst, 2000). In this book the Lanczos method is used, which is deceptively simple (Griffiths and Smith, 1991). Suppose the eigenproblem (3.18) has been reduced to finding eigenvalues and eigenvectors of a symmetric positive definite matrix [**A**] (the "Hermitian Eigenvalue Problem", Bai *et al.* 2000). In the Lanczos method, [**A**] is tridiagonalised by the following algorithm:

Choose a start vector $\{\mathbf{Y}_1\}$, and set $\{\mathbf{Y}_0\}$ to $\{\mathbf{0}\}$ and β_0 to 0, where $\{\boldsymbol{\alpha}\}$ and $\{\boldsymbol{\beta}\}$ are the leading diagonal and off-diagonal respectively of the tridiagonalisation.

Then for `j=1,neq` (the number of equations),

$$\begin{aligned}
\{\mathbf{V}\} &= [\mathbf{A}]\{\mathbf{Y}_1\} - \beta_{j-1}\{\mathbf{Y}_0\} \\
\{\mathbf{Y}_0\} &= \{\mathbf{Y}_1\} \\
\alpha_j &= \{\mathbf{Y}_1\}^{\mathrm{T}}\{\mathbf{V}\} \\
\{\mathbf{Z}\} &= \{\mathbf{V}\} - \alpha_j\{\mathbf{Y}_1\} \\
\beta_j &= \left(\{\mathbf{Z}\}^{\mathrm{T}}\{\mathbf{Z}\}\right)^{\frac{1}{2}} \\
\{\mathbf{Y}_1\} &= \{\mathbf{Z}\}/\beta_j
\end{aligned} \tag{3.30}$$

Again we see the basic algorithm involving a matrix–vector multiplication $[\mathbf{A}]\{\mathbf{Y}_1\}$ which can be carried out element-by-element following (3.23) together with whole array operations and inner products. However, for other than small systems, roundoff errors rapidly lead to erroneous results as orthogonality of the Lanczos vectors is lost. Practical algorithms (Bai *et al.*, 2000) involve rather elaborate techniques to re-orthogonalise these vectors after convergence to a given eigenvalue. Fortunately the basic structure of (3.30) is not affected, but the storage of information involving the previous vectors is a burden and can limit the applicability of some versions of the method for very large systems. The reduction of (3.18) to "standard form" is left until Section 3.9.2.

3.6 Incorporation of boundary conditions

Eigenvalues of stiffness matrices of freely floating elements or meshes are sometimes required, but normally in eigenvalue problems and always in equilibrium and propagation problems additional boundary information has to be supplied before solutions can be obtained. For example, the system matrix defined in Table 3.3 is singular and the set of equations (3.16) has no solution.

The simplest type of boundary condition occurs when the dependent variable in the solution is known to be zero at various points in the region (and hence nodes in the finite element mesh). When this occurs, the equation components associated with these degrees of freedoms are not required in the solution and information is given to the assembly routine which prevents these components from ever being assembled into the final system. Thus only the non-zero freedom values are solved for.

A variation of this condition occurs when the dependent variable has known, but non-zero, values at various locations (e.g. $\phi =$ constant). Although an elimination procedure could be devised, the way this condition is handled in practice is by adding a "large" number or "penalty" term, say 10^{20}, to the leading diagonal of the "stiffness" matrix in the row in which the prescribed value is required. The term in the same row of the right hand side vector is then set to the prescribed value multiplied by the augmented "stiffness" coefficient. For example, suppose the fluid head at node 5 in Figure 3.5 is known to be $\Phi_5 = 57.0$. The unconstrained set of equations (3.16) would be assembled, and the term $K_{5,5}$ augmented by adding 10^{20}. In the subsequent solution there would be an equation,

$$(K_{5,5} + 10^{20})\Phi_5 + \texttt{"small" terms} = 57.0\ (K_{5,5} + 10^{20}) \tag{3.31}$$

which would have the effect of making $\Phi_5 = 57.0$. Clearly this procedure is only successful if indeed "small terms" are small relative to 10^{20}.

This method could also be used to enforce the boundary condition $\Phi_i = 0.0$, and has some attractions in simplicity of data preparation.

Boundary conditions can also involve gradients of the unknown in the forms

$$\frac{\partial\phi}{\partial n} = 0 \tag{3.32}$$

$$\frac{\partial\phi}{\partial n} = C_1\phi \tag{3.33}$$

$$\frac{\partial\phi}{\partial n} = C_2 \tag{3.34}$$

where n is the normal to the boundary and C_1, C_2 are constants.

To be specific, consider a solution of the diffusion–advection equation (2.132) subject to boundary conditions (3.32), (3.33), and (3.34) respectively. When the second-order terms $c_x\partial^2\phi/\partial x^2$ and $c_y\partial^2\phi/\partial y^2$ are integrated by parts, boundary integrals of the type

$$\oint_S c_n\,[\mathbf{N}]^{\mathrm{T}}\,\frac{\partial\phi}{\partial n}\,l_n\,\mathrm{d}S \tag{3.35}$$

arise, where c_n is the diffusion property and l_n is the direction cosine of the normal to boundary S. Clearly the case $\partial\phi/\partial n = 0$ presents no difficulty, since the contour integral (3.35) vanishes and this is the default boundary condition obtained at any free surface of a finite element mesh.

However, (3.33) gives rise to an extra integral, which for the boundary element shown in Figure 3.7 is

$$\int_j^k c_y\,[\mathbf{N}]^{\mathrm{T}}\,C_1\,\tilde{\phi}\,l_y\,\mathrm{d}S \tag{3.36}$$

When $\tilde{\phi}$ is expanded as $[\mathbf{N}]\,\{\boldsymbol{\phi}\}$ we get an additional matrix,

$$\frac{-C_1c_y(x_k - x_j)}{6}\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.37}$$

which must be added to the left hand side of the element equations.

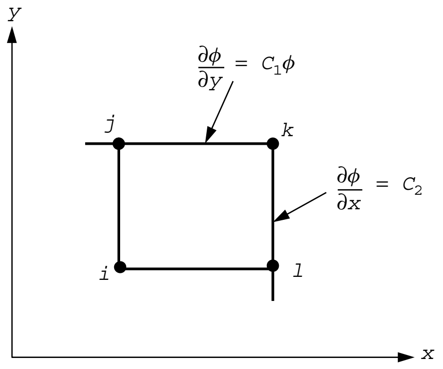

y
$\frac{\partial\phi}{\partial y} = C_1\phi$
j
k
$\frac{\partial\phi}{\partial x} = C_2$
i
l
x

Figure 3.7 Boundary conditions involving non-zero gradients of the unknown

For boundary condition (3.34) in Figure 3.7, the additional term is,

$$\int_k^l c_x\,[\mathbf{N}]^{\mathrm{T}}\,C_2\,l_x\,\mathrm{d}S \tag{3.38}$$

which is just a vector

$$\frac{C_2 c_x(y_k - y_l)}{2}\begin{Bmatrix}0\\0\\1\\1\end{Bmatrix} \tag{3.39}$$

which would be added to the right hand side of the element equations. For a further discussion of boundary conditions see Smith (1979).

In summary, boundary conditions of the type $\phi =$ constant or $\partial\phi/\partial n = 0$ are the most common and are easily handled in finite element analyses. The cases given by (3.33) and (3.34) in which $\partial\phi/\partial n$ is fixed to a non-zero value that is either a constant or a linear function of ϕ, are somewhat more complicated, but can be appropriately treated. Examples of the use of all these types of boundary specification are included in the applications Chapters 4 to 12.

3.7 Programming using building blocks

The programs in subsequent chapters are constituted from over 70 "building blocks" in the form of Fortran 95 functions and subroutines which perform the tasks of computing and integrating the element matrices, assembling these into system matrices if necessary and carrying out the appropriate equilibrium, eigenvalue or propagation calculations. In Chapter 12, the message passing interface MPI libraries handle the necessary communication between processors.

It is anticipated that users will elect to pre-compile all of the building blocks and to hold these permanently in a library. The library should then be automatically accessible to the calling programs by means of a simple `USE` statement at the beginning of the program (see Chapter 1, Section 1.9.9).

A summary of these subroutines and functions is given in Appendices D, E, and F where their actions and input/output parameters are described. Appendix D describes functions and subroutines that appear in the `main` library, and describes "black box" routines (concerned with some matrix operations), whose mode of action the reader need not necessarily know in detail, and special purpose routines which are the basis of specific finite element computations. Some of these routines should be thought of as an addition to the intrinsic Fortran 95 library functions such as `MATMUL` or `DOT_PRODUCT`, and could well be substituted with equivalents from a mathematical subroutine library, for example Basic Linear Algebra Subroutine (BLAS), perhaps tuned to a specific machine. Appendix E describes subroutines that appear in the `geom` library which holds customised routines, usually for generating element nodal coordinates and numbering for some of the simple geometries used with the specific examples described in this book. Appendix F describes the additional subroutines and functions needed by the Chapter 12 programs for their parallel algorithms.

3.7.1 Black box routines

Readers are reminded of the much improved array handling facilities of Fortran 95, compared with earlier FORTRANs. Chapter 1, Section 1.9 summarised such features as whole array operations and intrinsic array procedures, which mean that most simple array manipulations can be done using the power of the language itself and do not need to be user-supplied.

In the programs which follow from Chapter 4 onwards, only three simple functions or subroutines have been added to those provided as standard in the language. These are, `determinant`, `invert`, and `cross_product`

`determinant` returns the determinant of a 1×1, 2×2, or 3×3 matrix (usually the Jacobian matrix $[\mathbf{J}]$), `invert` computes the inverse of a (small) square matrix, again usually the Jacobian matrix (3.4) and `cross_product` computes the matrix result given by the cross-product of two vectors.

A second batch of subroutines shown in Table 3.4 is concerned with the solution of linear algebraic equations. The subroutines have been split into factorisation and forward/back-substitution phases.

Several subroutines are associated with eigenvalue and eigenvector determination; for example for symmetric banded matrices, `bandred` tridiagonalises the matrix and `bisect`

Table 3.4 Subroutines for solution of linear algebraic equations

Method	Gauss	Cholesky	Gauss	Gauss
Storage	Symmetric half-band	Symmetric skyline	Symmetric skyline	Unsymmetric full band
Factorisation	`banred`	`sparin`	`sparin_gauss`	`gauss_band`
Substitution	`bacsub`	`spabac`	`spabac_gauss`	`solve_band`

extracts all of the eigenvalues. It should be noted that these routines, although robust and accurate, can be inefficient both in storage requirements and in run-time and should not be used for solving very large problems, for which in any case it is unlikely that the full range of eigenmodes would be required. The various vector iteration methods (Bathe, 1996) should be resorted to in such cases.

One of the most effective of these is the Lanczos method (see Sections 3.5.5 and 3.9.2), in which subroutines `lancz1` and `lancz2` are used to calculate the eigenvalues and eigenvectors of a matrix.

When the Lanczos procedure is described in more detail, it will be found that, in common with its close relation the conjugate gradient procedure (Section 3.5.1), the method requires a matrix–vector product where the matrix is essentially the global system stiffness matrix, followed by a series of whole-vector operations. To save storage, the matrix–vector product can be done "element-by-element" and this feature is taken advantage of in Chapters 10 and 12.

Although simple matrix-by-vector multiplications can be accomplished by intrinsic procedure `MATMUL`, advantage is usually taken of the structure of global system matrix coefficients whenever possible. To allow for this, three special matrix-by-vector multiplication subroutines are provided as shown in Table 3.5 and further information on when these routines should be used is given in Sections 3.9, 3.10 and 3.12.

In a teaching text such as this, elaborate input and output procedures are avoided. It is expected that users may pre- and post-process their data using independent programs.

In order to describe the action of the remaining special purpose subroutines, (see Appendix D), it is necessary first to consider the properties of individual finite elements and then the representation of continua from assemblages of these elements. Static linear problems (including eigenproblems) are considered first. Thereafter modifications to programs to incorporate time dependence are added.

Table 3.5 Subroutines for matrix–vector multiplication

Storage	Symmetric skyline	Symmetric lower triangle	Unsymmetric full band
Assembly subroutine	`fsparv`	`formkb`	`formtb`
Matrix–vector multiplication subroutine	`linmul_sky`	`banmul`	`bantmul`

3.7.2 Special purpose routines

The job of these routines is to compute the element matrix coefficients, for example the "stiffness", to integrate these over the element area or volume and finally, if necessary, to assemble the element submatrices into a global system matrix or matrices. The black box routines for equation solution, eigenvalue determination and so on then take over to produce the final results. The remainder of this section introduces a notation that follows the variable names used in the subroutine listings. When appropriate, mnemonics are used so that the Jacobian matrix becomes `jac` and so on.

3.7.3 Plane elastic analysis using quadrilateral elements

As an example of element matrix calculation, consider the computation of the element stiffness matrix for plane elasticity given by (2.69)

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}] \, \mathrm{d}x \, \mathrm{d}y \tag{3.40}$$

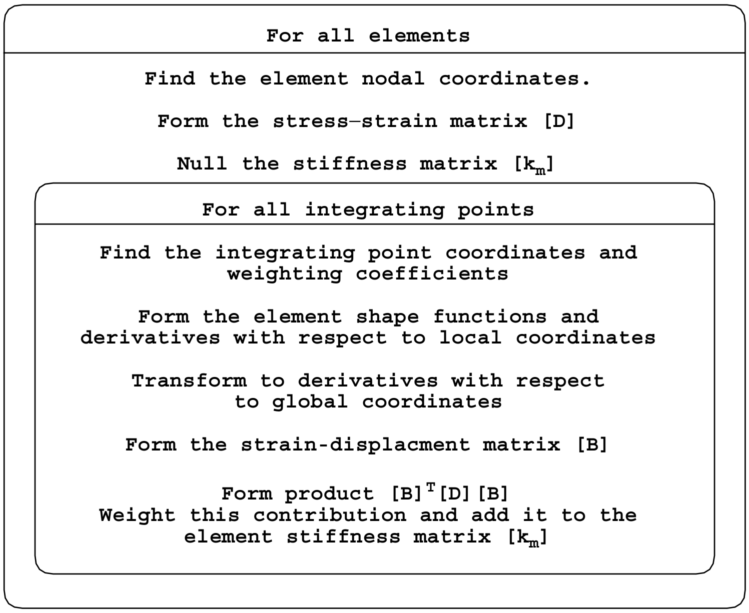
This formulation in the programs is described by the inner loop of the structure chart in Figure 3.8.

It is assumed for the moment that the element nodal coordinates (x, y) have been calculated and stored in the array `coord`. For example, for a 4-node quadrilateral (`nod=4`),

$$\texttt{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \tag{3.41}$$

The shape functions [**N**] are held in array `fun`, as specified in (3.1) by,

$$\texttt{fun} = \begin{Bmatrix} \frac{1}{4}(1-\xi)(1-\eta) \\ \frac{1}{4}(1-\xi)(1+\eta) \\ \frac{1}{4}(1+\xi)(1+\eta) \\ \frac{1}{4}(1+\xi)(1-\eta) \end{Bmatrix} \tag{3.42}$$



```
For all elements
    Find the element nodal coordinates.
    Form the stress-strain matrix [D]
    Null the stiffness matrix [k_m]
    For all integrating points
        Find the integrating point coordinates and
        weighting coefficients
        Form the element shape functions and
        derivatives with respect to local coordinates
        Transform to derivatives with respect
        to global coordinates
        Form the strain-displacment matrix [B]
        Form product [B]^T[D][B]
        Weight this contribution and add it to the
        element stiffness matrix [k_m]
```

Figure 3.8 Structure chart for element matrix assembly assuming numerical integration

The [**B**] matrix contains derivatives of the shape functions with respect to global coordinates, but first these are computed in the local coordinate system as

$$\texttt{der} = \begin{bmatrix} \dfrac{\partial\,\texttt{fun}^{\mathrm{T}}}{\partial\xi} \\ \dfrac{\partial\,\texttt{fun}^{\mathrm{T}}}{\partial\eta} \end{bmatrix} \tag{3.43}$$

or

$$\texttt{der} = \frac{1}{4}\begin{bmatrix} -(1-\eta) & -(1+\eta) & (1+\eta) & (1-\eta) \\ -(1-\xi) & (1-\xi) & (1+\xi) & -(1+\xi) \end{bmatrix} \tag{3.44}$$

The information in (3.42) and (3.44) for a 4-node quadrilateral (`nod=4`) is formed by the subroutines, `shape_fun` and `shape_der` for the specific Gaussian integration points $(\xi, \eta)_i$ held in the array `points` where i runs from 1 to `nip`, the total number of sampling points specified in each element. Figure 3.9(a) shows the node numbering and the order in which the sampling (Gauss) points are scanned in a "2-point" scheme. Since there are two integrating points in each coordinate direction `nip=4` in this example. In all cases, `points` and their corresponding `weights` (Table 3.1) are found by the subroutine `sample`, where `nip` can take the values 1, 4 or 9 for quadrilaterals.

The derivatives `der` must then be converted into their counterparts in the (x, y) coordinate system, `deriv`, by means of the Jacobian matrix transformation (3.4). From the isoparametric property (3.2),

$$\begin{aligned} x &= \frac{1}{4}(1-\xi)(1-\eta)x_1 + \frac{1}{4}(1-\xi)(1+\eta)x_2 \\ &\quad + \frac{1}{4}(1+\xi)(1+\eta)x_3 + \frac{1}{4}(1+\xi)(1-\eta)x_4 \\ y &= \frac{1}{4}(1-\xi)(1-\eta)y_1 + \frac{1}{4}(1-\xi)(1+\eta)y_2 \\ &\quad + \frac{1}{4}(1+\xi)(1+\eta)y_3 + \frac{1}{4}(1+\xi)(1-\eta)y_4 \end{aligned} \tag{3.45}$$

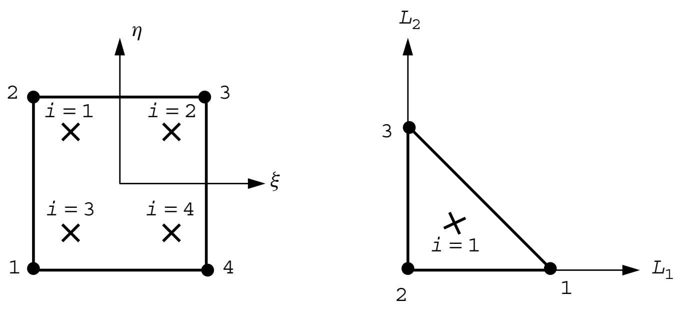


Figure 3.9 Integration schemes for (a) quadrilateral element with `nip=4`, and (b) triangular element with `nip=1`

and since the Jacobian matrix is given by

$$[\mathbf{J}] = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix} \tag{3.46}$$

it is clear that its terms can be obtained from (3.45), once the derivatives of the shape functions with respect to the local coordinates have been provided by subroutine `shape_der`, thus

```
CALL shape_der(der,points,i)
jac = MATMUL(der,coord)                 (3.47)
det = determinant(jac)
```

The function `determinant` computes `det`, the determinant of the Jacobian matrix, required later for the purposes of numerical integration.

In order to compute `deriv` we must invert `jac` using subroutine `invert` and finally carry out the multiplication of this inverse by `der` to give `deriv`,

```
CALL invert(jac)
deriv = MATMUL(jac,der)                 (3.48)
```

It should be noted that subroutine `invert` overwrites the original matrix by its inverse, thus `jac` in fact holds $[\mathbf{J}]^{-1}$ after the subroutine call.

The matrix $[\mathbf{B}]$ in (3.40) (called `bee` in program terminology) can now be assembled as it consists of components of `deriv`, and this operation is performed by the call,

```
CALL beemat(bee,deriv)                  (3.49)
```

The components of the integral of $[\mathbf{B}]^{\mathrm{T}}\,[\mathbf{D}]\,[\mathbf{B}]$, at each of the `nip` integrating points, can now be computed by transposing `bee` using the Fortran 95 intrinsic function `TRANSPOSE`, and by forming the stress–strain matrix `dee` using the subroutine `deemat`. In 2D analysis `(ndim=2)`, subroutine `deemat` gives the plane strain stress–strain matrix (2.70). The size of `dee` is given by `nst`, the number of components of stress and strain, which for 2D elastic analysis is equal to 3. A plane stress analysis would be obtained by simply replacing subroutine `deemat` by `fmdsig`.

The multiplication

```
btdb=MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)          (3.50)
```

gives `btdb`, the quantity to be integrated numerically by

$$\mathtt{km} = \sum_{i=1}^{\mathtt{nip}} \mathtt{det}_i * \mathtt{weights(i)} * \mathtt{btdb}_i \tag{3.51}$$

where `weights(i)` are the numerical integration weighting coefficients from (3.8).

As soon as the element matrix has been formed from (3.51) it can be assembled into the global system matrix (or matrices) by special subroutines described later in this chapter.

Following equation solution, once the global nodal displacements are known, the element displacements `eld` are retrieved, and the strains `eps` given by the strain-displacement relations,

$$\texttt{eps = MATMUL(bee,eld)} \tag{3.52}$$

where, in the case of a 4-node quadrilateral

$$\texttt{eld} = [u_1\ v_1\ u_2\ v_2\ u_3\ v_3\ u_4\ v_4]^{\mathrm{T}} \tag{3.53}$$

and stresses `sigma` from the stress–strain relations,

$$\texttt{sigma = MATMUL(dee,eps)} \tag{3.54}$$

The variables u and v are simply the nodal displacements in the x and y directions respectively assuming the nodal ordering of Figure 3.9(a).

In cases where the stiffness matrix `km` of a 4-node quadrilateral is required "analytically", the integration loop in Figure 3.8 is replaced by a single call to the subroutine `stiff4` (see Program 11.5). Similarly, when strains and stresses are back-calculated using 8-node quadrilateral elements (usually at the sampling points) the "analytical", subroutine `bee8` can be used to replace the lines of program given by (3.47) to (3.49) (see e.g. Programs 6.3, 6.8, and 6.9).

The shape functions and derivatives provided by subroutines `shape_fun` and `shape_der` allow analyses to be performed using quadrilateral elements with 4, 8 or 9 nodes (e.g. Program 5.1). A summary of the shape functions for all the elements used in this book is given in Appendix B.

Before describing the assembly process, which is common to all elements, modifications to the element matrix calculation for different situations will first be described.

3.7.4 Plane elastic analysis using triangular elements

The previous section showed how the stiffness matrix of a typical 4-node quadrilateral could be built up. In order to use triangular elements, very few alterations are required. For example, for a 3-node triangular element `(nod=3)`,

$$\texttt{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} \tag{3.55}$$

The shape functions [**N**] and their derivatives with respect to local coordinates at a particular location (L_1, L_2, L_3) (where $L_3 = 1 - L_1 - L_2$) are held in the arrays `fun` and `der`. Subroutine `shape_fun` delivers the shape functions,

$$\texttt{fun} = \begin{Bmatrix} L_1 \\ L_3 \\ L_2 \end{Bmatrix} \tag{3.56}$$

and subroutine `shape_der` delivers the derivatives with respect to L_1 and L_2

$$\texttt{der} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \tag{3.57}$$

The nodal numbering is shown in Figure 3.9(b). Exactly the same sequence of operations (3.47) to (3.51) as was used for quadrilaterals places the required derivatives with respect to (x, y) in `deriv`, finds the Jacobian determinant `det`, forms the `bee` matrix and numerically integrates the terms of the stiffness matrix `km`. For this simple element, only one integrating point at the element centroid is required (`nip=1`). For higher-order elements more triangular integrating points would be required. For example, the 6-node triangle would usually require `nip=3` for plane analysis. For integration over triangles, the sampling points in local coordinates (L_1, L_2) are held in the array `points` and the corresponding weighting coefficients in the array `weights`. As with quadrilaterals, both of these items are provided by the subroutine `sample`. This subroutine allows the total number of integrating points (`nip`) for triangles to take the values, 1, 3, 6, 7, 12, or 16. The coding should be referred to in order to determine the sequence in which the integrating points are sampled for `nip>1`.

3.7.5 Axisymmetric strain of elastic solids

The formation of the strain-displacement matrix follows a similar course to that described by (3.47) to (3.49), however in this case `bee` must be augmented by a fourth row corresponding to the "hoop" strain ϵ_θ as shown in (2.76). The cylindrical coordinates (r, z) replace their counterparts (x, y). The stress–strain matrix is given by equation (2.77) and is returned by subroutine `deemat` with `nst`, the number of stress and strain components now set to 4.

In this case, the integrated element stiffness is given by (2.74), namely

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}] \; r \, \mathrm{d}r \, \mathrm{d}z \tag{3.58}$$

where r is the radial coordinate given in the programs as `gc(1)` from the isoparametric relationship,

$$\texttt{gc = MATMUL(coord,fun)} \tag{3.59}$$

where `gc` and hence `fun` are evaluated at the sampling points.

The numerical integration summation in axisymmetry is written as

$$\texttt{km} = \sum_{i=1}^{\texttt{nip}} \texttt{det}_i \texttt{*weights(i)*btdb}_i \texttt{*gc(1)}_i \tag{3.60}$$

By comparison with (3.51) it may be seen that when evaluated numerically, the algorithms for axisymmetric and plane stiffness formation will be essentially the same, despite the fact that they are algebraically quite different. This is very significant from the points of view of programming effort and of program flexibility (e.g. Program 5.1).

However (3.58) now involves numerical evaluation of integrals involving $1/r$ (held in the [**B**] matrix, see 2.76) which do not have simple polynomial representations. Therefore, in contrast to plane problems, it will be impossible to evaluate (3.60) exactly by numerical means, and accuracy may deteriorate as r approaches zero. Provided integration points do not lie on the $r = 0$ axis, however, reasonable results are usually achieved using a similar order of quadrature to that used in plane analysis. Customised numerical integration schemes for axisymmetric elements are available (Griffiths, 1991), but are not used in this text.

3.7.6 Plane steady laminar fluid flow

It was shown in (2.126) that a fluid element has a "stiffness" or conductivity matrix defined in 2D by,

$$[\mathbf{k}_c] = \iint [\mathbf{T}]^{\mathrm{T}} [\mathbf{K}] [\mathbf{T}] \,\mathrm{d}x\,\mathrm{d}y \tag{3.61}$$

and the similarity to (3.40) is obvious. The matrix `deriv` simply contains the derivatives of the element shape functions with respect to (x, y) which were previously needed in the analysis of solids and formed by the sequence (3.47) to (3.48), while the constitutive matrix [**K**] (called `kay` in program terminology) contains the permeability (or conductivity) properties of the element in the form

$$\texttt{kay} = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix} \tag{3.62}$$

Numerical integration of the conductivity matrix in planar problems is completed by the sequence,

```
dtkd=MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)
```

$$\texttt{kc} = \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*dtkd}_i \tag{3.63}$$

By comparison with (3.51) it will be seen that these physically very different problems are likely to require similar solution algorithms.

3.7.7 Mass matrix formation

The mass matrix was shown in Chapter 2, for example (2.71), to take the general form

$$[\mathbf{m}_m] = \rho \iint [\mathbf{N}]^{\mathrm{T}} [\mathbf{N}] \,\mathrm{d}x\,\mathrm{d}y \tag{3.64}$$

where [**N**] holds the shape functions.

In the case of transient plane fluid flow, there is no density term, however with only one degree of freedom per node (nodof=1), the "mass" matrix mm is particularly easy to form by numerical integration giving the sequence

$$\texttt{CALL cross\_product(fun,fun,ntn)}$$

$$\texttt{mm} = \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*ntn}_i \tag{3.65}$$

where cross_product forms the array ntn prior to integration.

In the case of dynamic applications in plane stress or strain of solids (nodof=2), because of the arrangement of the displacement vector in (3.53) it is convenient to use a special subroutine ecmat to form the terms of the mass matrix as ecm before integration, hence,

$$\texttt{CALL ecmat(ecm,fun,ndof,nodof)}$$

$$\texttt{mm = rho*} \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*ecm}_i \tag{3.66}$$

where ndof is the number of degrees of freedom of the element and rho is the mass density of the material.

When "lumped" mass approximations are used mm becomes a diagonal matrix. For a 4-noded quadrilateral (nod=4), for example, the lumped mass matrix is given by

$$\texttt{mm = rho*area/nod*} [\mathbf{I}] \tag{3.67}$$

where area is the element area and [**I**] the unit matrix. For higher order elements, however, all nodes may not receive equal (and indeed intuitively "obvious") weighting. In Chapters 10 and 11, subroutine elmat is employed to generate the lumped mass matrix for 4- and 8-node quadrilaterals.

3.7.8 Higher-order 2D elements

The shape functions and derivatives provided by subroutines shape_fun and shape_der allow analyses to be performed using quadrilateral elements with 4, 8, or 9 nodes, or triangular elements with 3, 6, 10, or 15 nodes (e.g. Program 5.1). A summary of the shape functions for those elements is given in Appendix B. In the following two sections, examples of higher order quadrilateral and triangular elements are briefly described. As will be seen, programs using different element types will be identical, although operating on different sizes of arrays.

8-node quadrilateral

To emphasise the ease with which element types can be interchanged in programs, consider the next member of the isoparametric quadrilateral group, namely the 8-noded "quadratic"

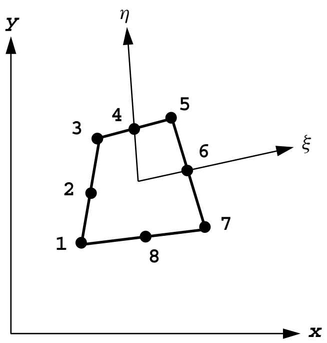


Figure 3.10 General quadratic quadrilateral element

quadrilateral element with mid-side nodes shown in Figure 3.10. The coordinate matrix becomes

$$\texttt{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_5 & y_5 \\ x_6 & y_6 \\ x_7 & y_7 \\ x_8 & y_8 \end{bmatrix} \tag{3.68}$$

and using the same local coordinate system for quadrilaterals, the shape functions $[\mathbf{N}]^{\mathrm{T}}$ are now

$$\texttt{fun} = \left\{ \begin{array}{c} \frac{1}{4}(1-\xi)(1-\eta)(-\xi-\eta-1) \\ \frac{1}{2}(1-\xi)(1-\eta^2) \\ \frac{1}{4}(1-\xi)(1+\eta)(-\xi+\eta-1) \\ \frac{1}{2}(1-\xi^2)(1+\eta) \\ \frac{1}{4}(1+\xi)(1+\eta)(\xi+\eta-1) \\ \frac{1}{2}(1+\xi)(1-\eta^2) \\ \frac{1}{4}(1+\xi)(1-\eta)(\xi-\eta-1) \\ \frac{1}{2}(1-\xi^2)(1-\eta) \end{array} \right\} \tag{3.69}$$

formed by subroutine `shape_fun`. The number of nodes (`nod=8`), and the dimensionality of the problem (`ndim=2`), serve to uniquely identify the required element, and hence the appropriate values of `fun`. Their derivatives with respect to local coordinates, `der`, are again formed by the subroutine `shape_der`.

The sequence of operations described by (3.47) to (3.51) obtains the terms needed for the element stiffness matrix integration.

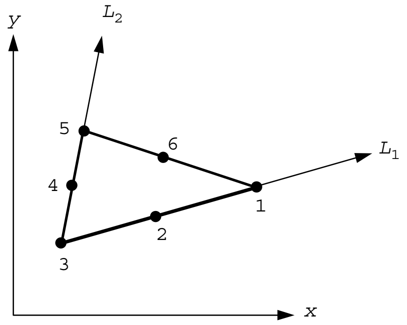


Figure 3.11 General 6-noded triangular element

6-node triangle

Another plane element available for use with the programs later in this book is the next member of the triangle family, namely the 6-noded triangular (`nod=6`) element as shown in Figure 3.11.

The coordinate matrix becomes

$$\texttt{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_5 & y_5 \\ x_6 & y_6 \end{bmatrix} \tag{3.70}$$

and using the same local coordinate system for triangles, the shape functions $[\mathbf{N}]^T$ are now

$$\texttt{fun} = \begin{Bmatrix} (2L_1 - 1)L_1 \\ 4L_3L_1 \\ (2L_3 - 1)L_3 \\ 4L_2L_3 \\ (2L_2 - 1)L_2 \\ 4L_1L_2 \end{Bmatrix} \tag{3.71}$$

Both the shape functions `fun` and the derivatives `der` are formed as usual by the subroutines `shape_fun` and `shape_der`. The sequence of operations described by (3.47) to (3.51) again follow to generate the stiffness matrix of the element.

3.7.9 Three-dimensional elements

Cuboidal elements

The shape functions and derivatives provided by subroutines `shape_fun` and `shape_der` allow analyses to be performed using cuboidal elements with 8, 14, or 20 nodes (see Appendix B).

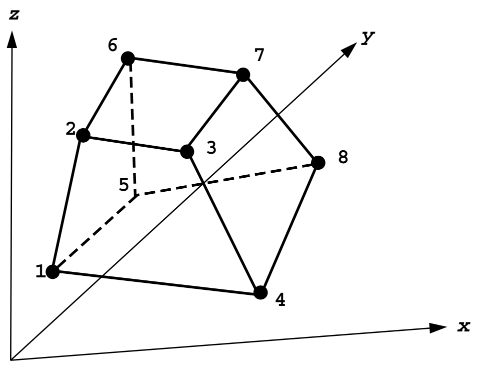


Figure 3.12 General linear hexahedron "brick" element

As was the case with changes of plane element types, changes of element dimensions are readily made. For example, the 8-node hexahedral "brick" element in Figure 3.12 is the three-dimensional extension of the 4-noded quadrilateral.

The coordinate matrix becomes

$$\texttt{coord} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \\ x_7 & y_7 & z_7 \\ x_8 & y_8 & z_8 \end{bmatrix} \tag{3.72}$$

and using the three-dimensional local coordinate system (ξ, η, ζ), the shape functions become

$$\texttt{fun} = \begin{Bmatrix} \frac{1}{8}(1-\xi)(1-\eta)(1-\zeta) \\ \frac{1}{8}(1-\xi)(1-\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1-\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1-\eta)(1-\zeta) \\ \frac{1}{8}(1-\xi)(1+\eta)(1-\zeta) \\ \frac{1}{8}(1-\xi)(1+\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1+\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1+\eta)(1-\zeta) \end{Bmatrix} \tag{3.73}$$

which together with their derivatives with respect to local coordinates, are as usual formed by the subroutines `shape_fun` and `shape_der` with `ndim=3` and `nod=8`.

The sequence of operations described by (3.47) to (3.48) results in deriv, the required gradients with respect to (x, y, z) and the Jacobian determinant det.

For a three-dimensional elastic solid the element stiffness is given by

$$[\mathbf{k}_m] = \iiint [\mathbf{B}]^{\mathrm{T}} [\mathbf{D}] [\mathbf{B}] \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \tag{3.74}$$

where bee and dee are formed by the subroutines beemat and deemat as usual, but with nst, the number of components of stress and strain now 6.

The numerical integration summation follows the same course as described previously for 2D elements in equations (3.51).

The 8-node cuboidal element will usually be integrated used "2-point" Gaussian integration (nip=8). For higher order cuboid elements the number of integrating points can expand rapidly. For example, "exact" integration of the 20-node cuboid element (see Appendix B) requires "3-point" integration, or nip=27. As with the 8-node plane element, "reduced" integration of the 20-node element (nip=8) can often improve its performance; however Smith and Kidger (1991) show that full nip=27 is essential with this element if spurious "zero energy" eigenmodes are to be avoided in the element stiffness. In addition to the conventional Gaussian rules, subroutine sample allows Irons's (1971) 14 and 15 point rules to be used for cuboid elements, and the reader is invited to experiment with these different integration strategies.

For 3D steady laminar fluid flow, the element conductivity or "stiffness" matrix is given by

$$[\mathbf{k}_c] = \iiint [\mathbf{T}]^{\mathrm{T}} [\mathbf{K}] [\mathbf{T}] \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \tag{3.75}$$

where the property matrix is formed as

$$[\mathbf{K}] = \begin{bmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{bmatrix} \tag{3.76}$$

Similarly, the "mass" matrix for fluid flow is

$$[\mathbf{m}_m] = \iiint [\mathbf{N}]^{\mathrm{T}} [\mathbf{N}] \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}z \tag{3.77}$$

In both cases, an identical sequence of operations as described previously for planar flow in equations (3.63) and (3.65) delivers the numerically integrated conductivity matrix kc and "mass" matrix mm.

A 14-node hexahedral element

The 20-node element mentioned previously is rather cumbersome and its stiffness can be expensive to compute in non-linear analyses, especially if employing nip=27. Furthermore, the 20-node brick element stiffness matrix exhibits "zero energy modes" when integrated using nip=8 (Smith and Kidger, 1991) which could be problematic.

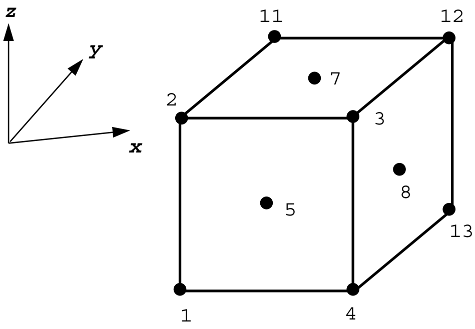


Figure 3.13 A 14-node hexahedron “brick” element

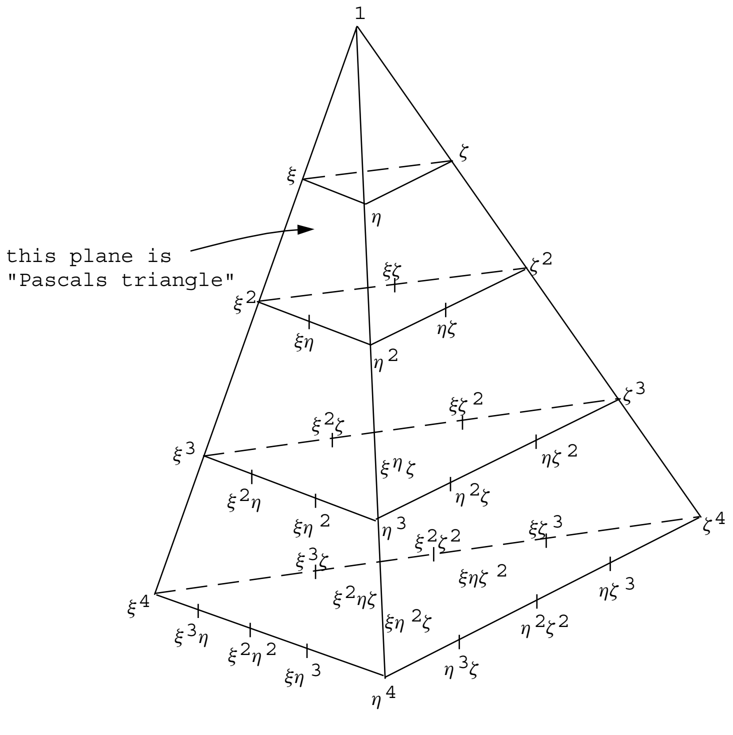


Figure 3.14 The Pascal pyramid

An alternative is to use a 14-node element (Smith and Kidger, 1992). As shown in Figure 3.13, this has 8 corner and 6 mid-face nodes. These nodes “populate” three-dimensional space more uniformly than 20 nodes do, since the latter are concentrated along the mesh lines. However, there is no unique choice of shape functions for a 14-node element. Figure 3.14 shows the “Pascal pyramid” of polynomials in (ξ, η, ζ) (or L_1, L_2, L_3) and one could experiment with various combinations of terms. It should be noted that the nearest plane of the pyramid is “Pascal’s triangle”, which can be used to select shape function terms for 2D elements. Smith and Kidger (1992) tried six permutations which were called “Types 1 to 6”. For example, Type 1 contained all 10 polynomials down to the second “plane” of the pyramid plus the terms $\xi\eta\zeta$, $\xi^2\eta$, $\eta^2\zeta$, and $\zeta^2\xi$ from the third “plane”. Type 6 selectively contained terms as far down as the fifth “plane” and this is the version available in library subroutines `shape_fun` and `shape_der`. Computer algebra was essential when deriving the shape functions for the “Type 6” element which are listed in Appendix B.

Tetrahedral elements

An alternative element for 3D analysis is the tetrahedron, the simplest of which has 4 corner nodes and is called the “constant strain” tetrahedron. The local coordinate system involves mapping a general tetrahedron onto a right-angled tetrahedron with three orthogonal sides of unit length as shown in Figure 3.15 and Appendix B. This approach can be shown to be identical to “volume coordinates”. For example, point P can be identified uniquely by the coordinates (L_1, L_2, L_3). As with triangles, an additional coordinate L_4 given by

$$L_4 = 1 - L_1 - L_2 - L_3 \tag{3.78}$$

is sometimes retained for algebraic convenience.

The shape functions for the “constant strain” tetrahedron are

$$\texttt{fun} = \begin{Bmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \end{Bmatrix} \tag{3.79}$$

and these, together with their derivatives with respect to L_1, L_2 and L_3 are formed by the usual subroutines `shape_fun` and `shape_der` with `ndim=3` and `nod=4`. The sequence

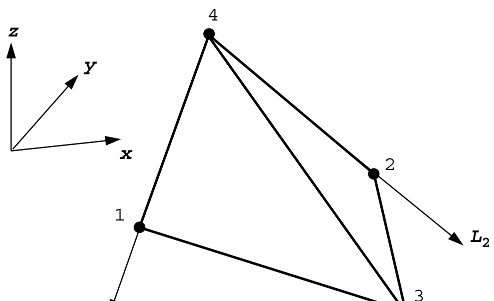


Figure 3.15 A 4-node tetrahedron element

of operations described by (3.47) to (3.51) again follows to generate the stiffness matrix of the element.

The "constant strain" tetrahedron requires only one integrating point (`nip=1`) situated at the element centroid.

The addition of mid-side nodes results in the 10-node tetrahedron, which represents the next member of family. Reference to Figure 3.14 shows that the tetrahedral family of shape functions maps naturally onto the Pascal pyramid (4, 10, 20, 35, etc. nodes). These elements could easily be implemented by the interested reader.

Transient, coupled poro-elastic transient and elastic-plastic analysis all involve manipulations of the few simple element property matrices described above. Before describing such applications, methods of assembling elements and of solving linear equilibrium and eigenvalue problems will first be discussed.

3.7.10 Assembly of elements

The special purpose subroutines `formnf`, `formkb`, `formku`, `fkdiag`, `formtb`, and `fsparv` are concerned with assembling the individual element matrices to form the global matrices that approximate the desired continuum, if assembly is preferred to an "element-by-element" approach. Allied to these there must be a specification of the geometrical details, in particular the nodal coordinates of each element and the element's place in some overall node numbering scheme.

Large finite element programs contain mesh generation code, which is usually of some considerable complexity. Indeed, in much finite element work, the most expensive and time consuming task is the preparation of the input data using the mesh generation routines. In the present book, this aspect of the computations is essentially ignored and most examples are restricted to simple classes of geometry, such as those shown in Figure 3.16, which can be automatically built up by "geometry" subroutines.

Alternatively, more general purpose programs are presented later in which the element geometries and nodal connectivities are simply read into the analysis program as data, having previously been worked out by an independent mesh generator.

In the present work, a typical program might use plane 4-node rectangular elements, so subroutines such as `geom_rect` are provided to generate coordinates and node numbering.

A full list of "geometry" subroutines is given in Appendix E.

With reference to Figure 3.16, the nodes of the mesh are first assigned numbers as economically as possible (i.e. always numbering in the "shorter" direction to minimise the bandwidth). Associated with each node are degrees of freedom (displacements, fluid potentials, etc.) which are numbered in the same order as the nodes. However, account can be taken at this stage of whether a degree of freedom exists or whether, generally at the boundaries of the region, the freedom is suppressed, in which case that freedom number is assigned the value zero. Alternatively, all freedoms can be assigned values whether they equal zero or not, and fixed later to their required values using the "penalty" approach. This latter approach leads to larger systems of equations, but with simpler freedom numbering.

In the examples that follow, the "zero freedoms" have been removed from the assembly process.

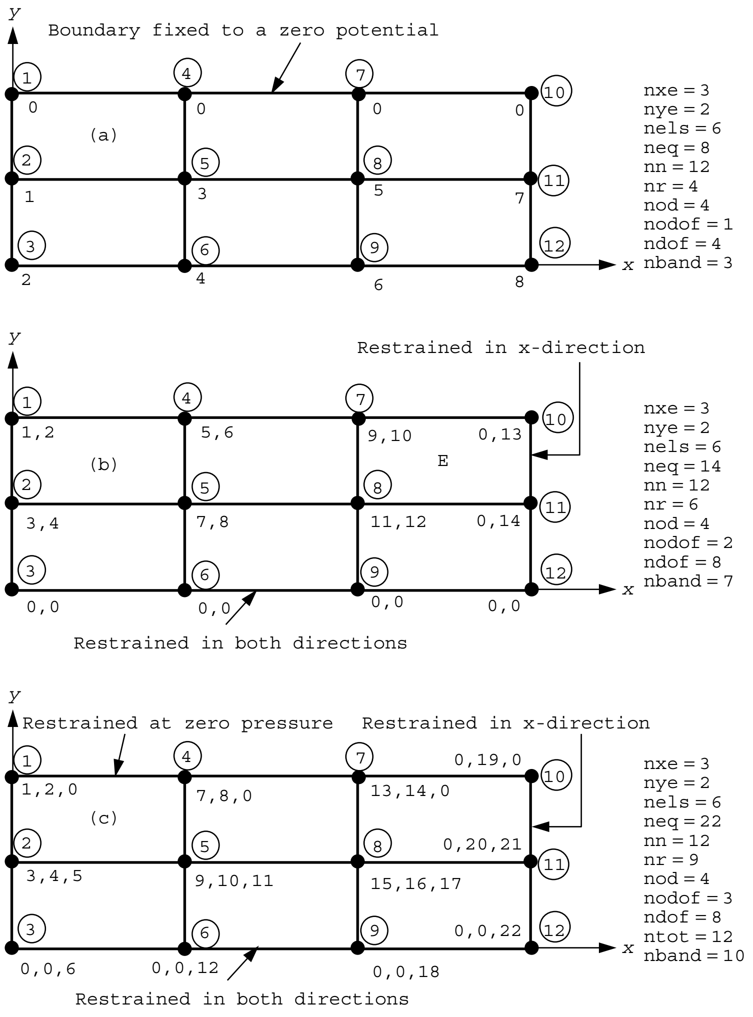


Figure 3.16 Numbering system and data for regular meshes. (a) One degree of freedom per node. (b) Two degrees of freedom per node. (c) Coupled problem with three degrees of freedom per node

The variables in Figure 3.16 have the following meaning:

nxe	elements counting in x direction
nye	elements counting in y direction
nels	total number of elements
neq	total number of (non-zero) freedoms in problem
nn	total number of nodes in problem
nr	number of restrained nodes
nod	number of nodes per element

`nodof`	number of freedoms per node
`ndof`	number of freedoms per element
`ntot`	total number of freedoms per element (for coupled problems)
`nband`	the half-bandwidth

In many of the programs in the book which use the geometry subroutine `geom_rect`, the values of `nels` and `nn` are first calculated by the subroutine `mesh_size`.

In scalar potential problems, there is one degree of freedom possible per node, the "potential" ϕ (Figure 3.16(a)). In plane or axisymmetric strain problems there are two, namely u and v, the components of displacement in the x- and y- (or r- and z-) directions respectively (Figure 3.16(b)). In planar coupled solid-fluid problems there are three, with u, v, and u_w (where u_w = excess pressure, Figures 3.16(c)), and similarly in Navier–Stokes applications the order is u, p, v (where p = pressure, and u and v represent velocity components). Regular 3D displacement problems have three freedoms per node given by u, v, and w (in x, y, and z). For 3D coupled solid–fluid problems, the 4 degrees of freedom per node are u, v, w, and u_w, while for 3D Navier–Stokes the order is u, p, v, and w.

The information about the degrees of freedom associated with any node in specific problems is stored in an integer array `nf` called the "node freedom array", formed by the subroutine `formnf`.

The node freedom array `nf` has `nodof` rows, one for each degree of freedom per node, and `nn` columns, one for each node in the problem analysed. Formation of `nf` is achieved by specifying, as data to be read in, the number of any node which has one or more restrained freedoms, followed by the digit 0 if the node is restrained in that sense and by the digit 1 if it is not. The appropriate Fortran 95 coding is

```
READ(10,*)nr,(k,nf(:,k),i=1,nr)

CALL formnf(nf)
```

For example, to create `nf` for the problem shown in Figure 3.16(b) the data specified and the resulting `nf` are listed in Table 3.6.

In regular rectangular meshes, data for generating the mesh coordinates and connectivity depends on a "geometry" subroutine such as `geom_rect`, which takes as input the number of elements in the $x(r)$- and $y(z)$-directions respectively (`nxe`, `nye`), together with the x- and y-coordinates of the vertical and horizontal lines that form the mesh (held in

Table 3.6 Formation of a typical nodal freedom array

Data			Resulting `nf` array											
6			1	3	0	5	7	0	9	11	0	0	0	0
3	0	0	2	4	0	6	8	0	10	12	0	13	14	0
6	0	0												
9	0	0												
10	0	1												
11	0	1												
12	0	0												

vectors `x_coords` and `y_coords`). For each element, the subroutine works out the nodal coordinates (held in array `coord`) and the nodal numbering (held in vector `num`). Both the coordinates and the node numbering are generated in an order consistent with the local node numbering of the element. In the case of a 4-node quadrilateral, this would be in the order 1-2-3-4 as shown in Figure 3.1 (see Appendix B for local numbering of all elements used in this book). For example, element E in Figure 3.16(b) has the node numbering vector

$$\texttt{num} = [8 \;\; 7 \;\; 10 \;\; 11]^T \tag{3.80}$$

Once the element node numbering is found, the "steering vector" `g` which holds the freedom numbers for the element can be found by comparing `num` with the "node freedom array" `nf`. This operation is performed by the subroutine `num_to_g`, which, again for element E would give

$$\texttt{g} = [11 \;\; 12 \;\; 9 \;\; 10 \;\; 0 \;\; 13 \;\; 0 \;\; 14]^T \tag{3.81}$$

In its turn `g` is used to assemble the coefficients of the element property matrices such as `km`, `kc`, and `mm` into the appropriate places in the overall global coefficient matrix. This is done according to one of the schemes given in Table 3.7.

A simple three-dimensional mesh is shown in Figure 3.17 and the system coefficients can again be assembled using the same building blocks.

Although the user of these subroutines does not strictly need to know how the storage is carried out, examples are given in Figure 3.18 of the most commonly used storage strategies generated by the subroutines in Table 3.7.

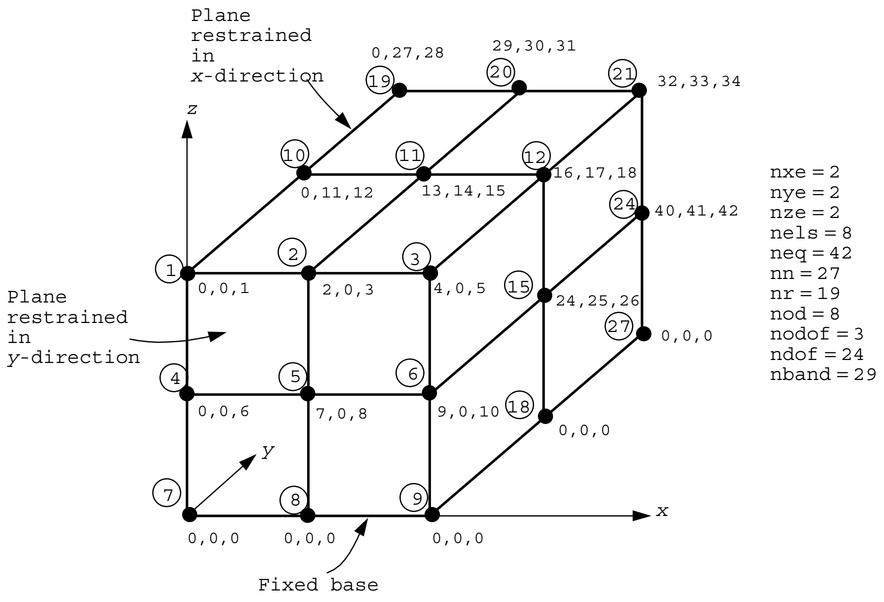
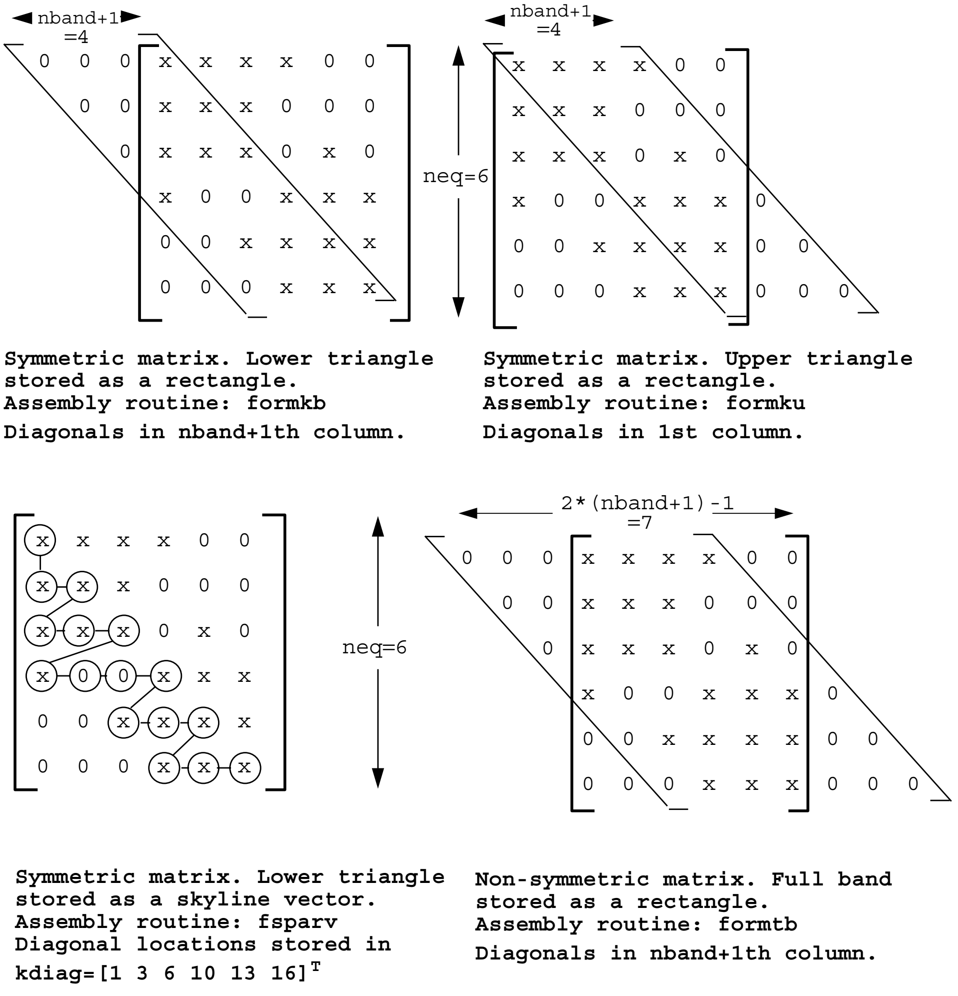


Figure 3.17 Numbering system and data for a regular 3D mesh with three degrees of freedom per node

Table 3.7 Summary of assembly subroutines

Subroutine	Banding/symmetry/ triangle?	Storage	Diagonals
`fsparv`	Yes/yes/lower	`kv(1:kdiag(neq))`	`kdiag` "skyline"
`formku`	Yes/yes/upper	`ku(neq,nband+1))`	`1`st col.
`formkb`	Yes/yes/lower	`kb(neq,nband+1))`	`nband+1`th col.
`formtb`	Yes/no/both	`pb(neq,2(nband+1)-1)`	`nband+1`th col.



Symmetric matrix. Lower triangle stored as a rectangle.
Assembly routine: formkb
Diagonals in nband+1th column.

Symmetric matrix. Upper triangle stored as a rectangle.
Assembly routine: formku
Diagonals in 1st column.

Symmetric matrix. Lower triangle stored as a skyline vector.
Assembly routine: fsparv
Diagonal locations stored in
kdiag=[1 3 6 10 13 16]^T

Non-symmetric matrix. Full band stored as a rectangle.
Assembly routine: formtb
Diagonals in nband+1th column.

Figure 3.18 Examples of storage strategies and assembly subroutines for symmetric and non-symmetric banded arrays (`neq=6`, `nband=3`)

The strategy in the majority of programs in this book is the “skyline” approach. It should be noted that `fsparv` stores only those numbers within the “skyline” in the form of a vector. Information regarding the position of the diagonal terms within the resulting vector is held in the integer vector `kdiag` formed by subroutine `fkdiag`.

3.8 Solution of equilibrium equations

If an assembly strategy is chosen, after specification of boundary conditions, a typical global equilibrium equation is

$$[\mathbf{K}_m]\{\mathbf{U}\} = \{\mathbf{F}\} \tag{3.82}$$

in which the terms of the global coefficient matrix $[\mathbf{K}_m]$ have usually been assembled by subroutine `fsparv` and stored in a vector called `kv`, and the global right-hand side vector $\{\mathbf{F}\}$, usually stored in a vector called `loads`, is just input as data. The black box subroutines for equation solution for the unknown vector $\{\mathbf{U}\}$ depend of course on the method of coefficient storage according to the schemes shown in Table 3.8.

Table 3.8 Equation solution subroutines

Coefficients formed by	Solution routines	Method
`fsparv`	`sparin` `spabac`	Cholesky
`fsparv`	`sparin_gauss` `spabac_gauss`	Gauss
`formtb`	`gauss_band` `solve_band`	Gauss

In fact, to save storage, all of the solution routines overwrite the right hand side by the solution. That is, following solution of (3.82) the vector `loads` holds the solution. The storage strategy adopted by the compiler and the hardware, as described in Chapter 1, strongly influences the solution method that should be used for large problems.

For very large systems, assembly would not be used at all, and the iterative processes described in Section 3.5 would be substituted.

3.9 Evaluation of eigenvalues and eigenvectors

Before solution, it is often necessary to reduce the eigenvalue equation to “standard form”

$$[\mathbf{A}]\{\mathbf{Z}\} = \omega^2\{\mathbf{Z}\} \tag{3.83}$$

where $[\mathbf{A}]$ is symmetric, $\{\mathbf{Z}\}$ represents the eigenvector and ω^2 the eigenvalue.

3.9.1 Jacobi algorithm

The problem frequently presents itself in the form of a generalised eigenproblem of the form

$$[\mathbf{K}_m]\,\{\mathbf{X}\} = \omega^2\,[\mathbf{M}_m]\,\{\mathbf{X}\} \tag{3.84}$$

where $[\mathbf{K}_m]$ and $[\mathbf{M}_m]$ are the global stiffness and mass matrices respectively. For example, the stiffness can be stored as a banded matrix having been formed by subroutine `formku` (Table 3.7). The mass would then be either a banded matrix with the same structure as the stiffness or more frequently if the mass is assumed to be "lumped", a diagonal matrix which can be stored in a vector `diag` with the help of subroutine `formlump`. In order to reduce (3.84) to the required form (3.83) it is necessary to factorise the mass matrix by forming

$$[\mathbf{M}_m] = [\mathbf{L}]\,[\mathbf{L}]^{\mathrm{T}} \tag{3.85}$$

While this is essentially a Cholesky factorisation, it is particularly simple in the case of a diagonal matrix, in which case the diagonal terms in $[\mathbf{L}]$ are simply the square roots of the diagonal terms in $[\mathbf{M}_m]$ and the inverse of $[\mathbf{L}]$ merely consists of the reciprocals of these square roots. In any case,

$$[\mathbf{K}_m]\,\{\mathbf{X}\} = \omega^2\,[\mathbf{L}]\,[\mathbf{L}]^{\mathrm{T}}\,\{\mathbf{X}\} \tag{3.86}$$

which is then reduced to standard form by making the substitution

$$[\mathbf{L}]^{\mathrm{T}}\,\{\mathbf{X}\} = \{\mathbf{Z}\} \tag{3.87}$$

Then

$$[\mathbf{L}]^{-1}\,[\mathbf{K}_m]\,[\mathbf{L}]^{-T}\,\{\mathbf{Z}\} = \omega^2\,\{\mathbf{Z}\} \tag{3.88}$$

is of the desired form (3.83). Having solved for $\{\mathbf{Z}\}$, the required eigenvectors $\{\mathbf{X}\}$ are readily recovered using (3.87). The subroutines `bandred` and `bisect` deliver the appropriate eigenvalues.

3.9.2 Lanczos algorithm

The transformation technique previously described is robust in that it will not fail to find an eigenvalue or to detect multiple roots. However, it is expensive to use on large problems for which, in general, vector iteration methods are preferable. Since the heart of all of these involves a matrix-by-vector product as shown in Section 3.5.5, they are ideal for "element-by-element" manipulation, in which case the lumped mass matrix is best used. Programs described in Chapters 10 and 12 use the HSL (2002) package EA25 which implements the work of Parlett and Reid (1981) on the Lanczos algorithm. These subroutines calculate the eigenvalues and eigenvectors of a symmetric matrix, say $[\mathbf{A}]$ from (3.83), requiring the user only to compute matrix–vector products and whole vector additions of the form $[\mathbf{A}]\{\mathbf{V}\} +$

$\{\mathbf{U}\}$ similar to those described in (3.30) for $\{\mathbf{U}\}$ and $\{\mathbf{V}\}$ provided by the subroutines. These operations can be carried out element-wise as was described in Section 3.5.

There is a slight additional complexity in that, as in (3.84) we often have the generalised eigenvalue problem to solve. For consistent mass approximations $[\mathbf{L}][\mathbf{L}]^{\mathrm{T}}$ in (3.85) is formed by Cholesky factorisation using `cholin`. Then on each Lanczos step from (3.30), wherever $[\mathbf{L}]^{-1}[\mathbf{K}_m][\mathbf{L}]^{-T}\{\mathbf{V}\}+\{\mathbf{U}\}$ is called for, we compute

$$\begin{aligned}
[\mathbf{L}]^{\mathrm{T}}\{\mathbf{Z}_1\} &= \{\mathbf{V}\} && \texttt{chobk2}\\
\{\mathbf{Z}_2\} &= [\mathbf{K}_m]\{\mathbf{Z}_1\} && \texttt{banmul}\\
[\mathbf{L}]\{\mathbf{Z}_3\} &= \{\mathbf{Z}_2\} && \texttt{chobk1}\\
\{\mathbf{U}\} &= \{\mathbf{U}\}+\{\mathbf{Z}_3\} && \text{whole vector operation}
\end{aligned} \tag{3.89}$$

Finally, the true eigenvectors are recovered from the transformed ones using backward substitution (`chobk2` as in 3.87). For lumped mass approximations, essentially the same operations are performed but the diagonal nature of $[\mathbf{L}]$ means that subroutine calls can be dispensed with.

3.10 Solution of first order time dependent problems

A typical equation at the element level from (2.131) is given by

$$[\mathbf{k}_c]\{\boldsymbol{\phi}\}+[\mathbf{m}_m]\left\{\frac{\mathrm{d}\boldsymbol{\phi}}{\mathrm{d}t}\right\}=\{\mathbf{q}\} \tag{3.90}$$

where $\{\boldsymbol{\phi}\}$ represents the dependent variable, and $\{\mathbf{q}\}$ represents any additional sources or sinks, and may be a function of time. There are many ways of integrating this set of ordinary differential equations, and modern methods for small numbers of equations would probably be based on variable order, variable timestep methods with error control. However, for large engineering systems more primitive methods are still mainly used, involving linear interpolations and fixed time steps Δt. If an element assembly method is to be used, $[\mathbf{k}_c]$ becomes $[\mathbf{K}_c]$, $[\mathbf{m}_m]$ becomes $[\mathbf{M}_m]$ and the basic equations can be written at two consecutive time steps "0" and "1" as follows:

$$[\mathbf{K}_c]\{\boldsymbol{\Phi}\}_0+[\mathbf{M}_m]\left\{\frac{\mathrm{d}\boldsymbol{\Phi}}{\mathrm{d}t}\right\}_0=\{\mathbf{Q}\}_0 \tag{3.91}$$

$$[\mathbf{K}_c]\{\boldsymbol{\Phi}\}_1+[\mathbf{M}_m]\left\{\frac{\mathrm{d}\boldsymbol{\Phi}}{\mathrm{d}t}\right\}_1=\{\mathbf{Q}\}_1 \tag{3.92}$$

where $\{\boldsymbol{\Phi}\}$ and $\{\mathbf{Q}\}$ represent the global counterparts of $\{\boldsymbol{\phi}\}$ and $\{\mathbf{q}\}$.

A third equation advances the solution from "0" to "1" using a weighted average of the gradients at the beginning and end of the time interval, thus,

$$\{\boldsymbol{\Phi}\}_1=\{\boldsymbol{\Phi}\}_0+\Delta t\left((1-\theta)\left\{\frac{\mathrm{d}\boldsymbol{\Phi}}{\mathrm{d}t}\right\}_0+\theta\left\{\frac{\mathrm{d}\boldsymbol{\Phi}}{\mathrm{d}t}\right\}_1\right) \tag{3.93}$$

Elimination of $\{\mathrm{d}\boldsymbol{\Phi}/\mathrm{d}t\}_0$ and $\{\mathrm{d}\boldsymbol{\Phi}/\mathrm{d}t\}_1$ from equations (3.91) to (3.93) leads to the following recurrence equation between time steps “0” and “1”:

$$([\mathbf{M}_m] + \theta\Delta t\,[\mathbf{K}_c])\,\{\boldsymbol{\Phi}\}_1 = ([\mathbf{M}_m] - (1-\theta)\Delta t\,[\mathbf{K}_c])\,\{\boldsymbol{\Phi}\}_0 + \theta\Delta t\,\{\mathbf{Q}\}_1 + (1-\theta)\Delta t\,\{\mathbf{Q}\}_0 \tag{3.94}$$

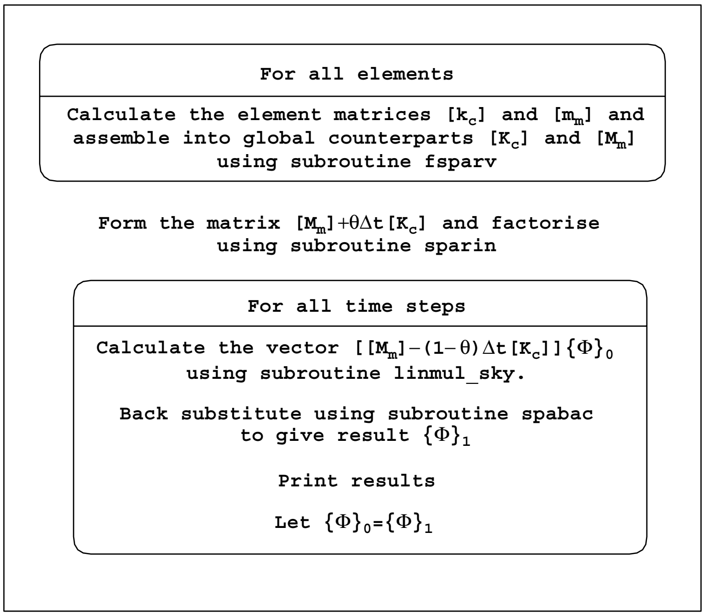
This system is only unconditionally “stable” (i.e. errors will not grow unboundedly) if $\theta \geq 1/2$. Common choices would be $\theta = 1/2$, giving the “Crank–Nicolson” method, where, assuming for the moment $\{\mathbf{Q}\} = \{\mathbf{0}\}$,

$$\left([\mathbf{M}_m] + \frac{\Delta t}{2}\,[\mathbf{K}_c]\right)\{\boldsymbol{\Phi}\}_1 = \left([\mathbf{M}_m] - \frac{\Delta t}{2}\,[\mathbf{K}_c]\right)\{\boldsymbol{\Phi}\}_0 \tag{3.95}$$

and $\theta = 1$ giving the “fully implicit” method,

$$([\mathbf{M}_m] + \Delta t\,[\mathbf{K}_c])\,\{\boldsymbol{\Phi}\}_1 = [\mathbf{M}_m]\,\{\boldsymbol{\Phi}\}_0 \tag{3.96}$$

In programming terms, the marching process from “0” to “1” involves first, a matrix-by-vector multiplication on the right hand side of (3.95) or (3.96) using `linmul_sky` (assuming subroutine `fsparv` has been used for assembly), and second, solution of a set of linear equations on each timestep. If the $[\mathbf{K}_c]$ and $[\mathbf{M}_m]$ matrices do not change with time, the factorisation of the left hand side coefficients, which is a time consuming operation, need only be performed once, as shown in the structure chart in Figure 3.19.



```
For all elements
    Calculate the element matrices [k_c] and [m_m] and
    assemble into global counterparts [K_c] and [M_m]
    using subroutine fsparv

Form the matrix [M_m]+θΔt[K_c] and factorise
using subroutine sparin

For all time steps
    Calculate the vector [[M_m]-(1-θ)Δt[K_c]]{Φ}_0
    using subroutine linmul_sky.

    Back substitute using subroutine spabac
    to give result {Φ}_1

    Print results

    Let {Φ}_0={Φ}_1
```

Figure 3.19 Structure chart for first-order time dependent problems by implicit methods using an assembly strategy

The "implicit" strategies described above are quite effective for linear problems (constant $[\mathbf{K}_c]$ and $[\mathbf{M}_m]$), however storage requirements can be considerable, and in non-linear problems the necessity to refactorise $([\mathbf{M}_m] + \theta\Delta t\,[\mathbf{K}_c])$ can lead to lengthy calculations.

Storage can be saved by replacing subroutine `linmul_sky` by an element-by-element matrix–vector multiply, and by solving the simultaneous equations iteratively (e.g. by pcg). Such a strategy can be attractive for parallel processing and examples are given in Chapter 12.

There is another alternative, widely used for second-order problems (see also Section 3.13.4), in which θ is set to zero and the $[\mathbf{M}_m]$ matrix is "lumped" (see equation 3.67). In this "explicit" approach, the system to be solved is

$$[\mathbf{M}_m]\{\boldsymbol{\Phi}\}_1 = ([\mathbf{M}_m] - \Delta t\,[\mathbf{K}_c])\{\boldsymbol{\Phi}\}_0 \tag{3.97}$$

or

$$\{\boldsymbol{\Phi}\}_1 = [\mathbf{M}_m]^{-1}([\mathbf{M}_m] - \Delta t\,[\mathbf{K}_c])\{\boldsymbol{\Phi}\}_0 \tag{3.98}$$

Although written here at the global level, in the case of $\theta = 0$ no global matrix assembly is needed because the matrix–vector products on the right hand side of equation (3.98) can all be achieved using an element-by-element strategy involving manipulations of the element matrices $[\mathbf{k}_c]$ and $[\mathbf{m}_m]$.

Although the "explicit" algorithm is simple, the disadvantage is that (3.98) is only stable on condition that Δt is "small", and in practice perhaps so small that real times of interest would require an excessive number of steps.

Yet another element-by-element approach which conserves computer storage while preserving the stability properties of "implicit methods" involves "operator splitting" on an element-by-element product basis (Hughes *et al.*, 1983; Smith *et al.*, 1989). Although not necessary for the operation of the method, the simplest algorithms result from "lumping" $[\mathbf{M}_m]$. Assuming again that $\{\mathbf{Q}\} = \{\mathbf{0}\}$, equation (3.94) can be written,

$$\{\boldsymbol{\Phi}\}_1 = ([\mathbf{M}_m] + \theta\Delta t\,[\mathbf{K}_c])^{-1}([\mathbf{M}_m] - (1-\theta)\Delta t\,[\mathbf{K}_c])\{\boldsymbol{\Phi}\}_0 \tag{3.99}$$

The element-by-element "operator splitting" methods are based on binomial theorem expansions of $([\mathbf{M}_m] + \theta\Delta t\,[\mathbf{K}_c])^{-1}$ which neglect product terms. When $[\mathbf{M}_m]$ is "lumped" (diagonal), the method is particularly straightforward because $[\mathbf{M}_m]$ can effectively be replaced by $[\mathbf{I}]$, the unit matrix, where

$$([\mathbf{I}] + \theta\Delta t\,[\mathbf{K}_c])^{-1} = \left([\mathbf{I}] + \theta\Delta t \sum [\mathbf{k}_c]\right)^{-1} \tag{3.100}$$

$$\approx \prod ([\mathbf{I}] + \theta\Delta t\,[\mathbf{k}_c])^{-1} \tag{3.101}$$

where $\sum$ indicates a summation, and $\prod$ indicates a product over all the elements. As was the case with implicit methods, optimal accuracy consistent with stability is achieved for $\theta = 1/2$. It is shown by Hughes *et al.* (1983) that further optimisation is achieved by splitting further to

$$[\mathbf{k}_c] = \frac{1}{2}[\mathbf{k}_c] + \frac{1}{2}[\mathbf{k}_c] \tag{3.102}$$

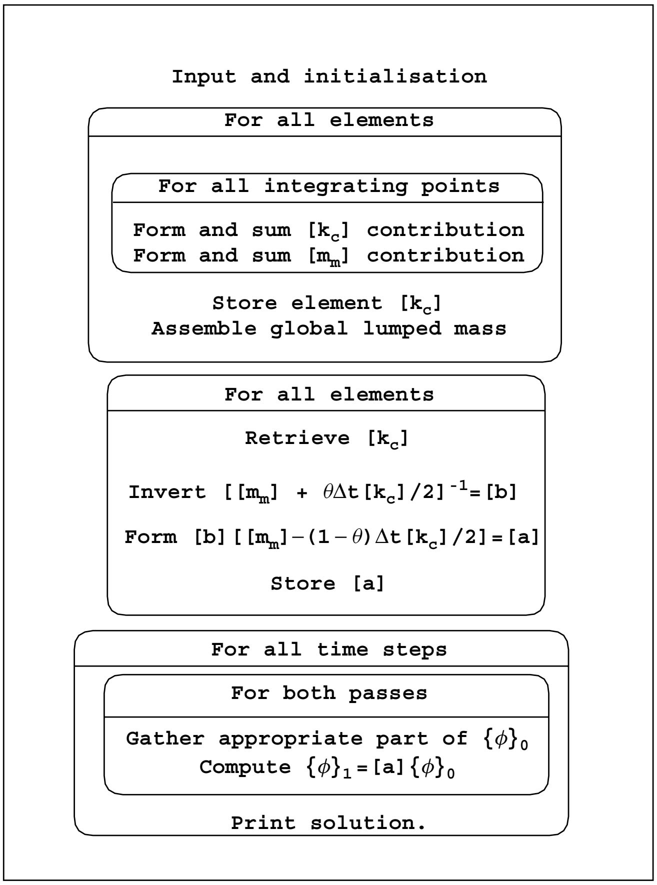

Input and initialisation
For all elements
For all integrating points
Form and sum $[k_c]$ contribution
Form and sum $[m_m]$ contribution
Store element $[k_c]$
Assemble global lumped mass
For all elements
Retrieve $[k_c]$
Invert $[[m_m] + \theta\Delta t[k_c]/2]^{-1} = [b]$
Form $[b][[m_m] - (1-\theta)\Delta t[k_c]/2] = [a]$
Store [a]
For all time steps
For both passes
Gather appropriate part of $\{\phi\}_0$
Compute $\{\phi\}_1 = [a]\{\phi\}_0$
Print solution.

Figure 3.20 Structure chart for the element-by-element product algorithm (two pass)

and carrying out the product (3.101) by sweeping twice through the elements. They suggest from first to last and back again, but clearly various choices of sweeps could be employed. It can be shown that as $\Delta t \rightarrow 0$, any of these processes converges to the true solution of the global problem. A structure chart for the process is shown in Figure 3.20 and examples of all the methods described in this section are implemented in Chapter 8. Consistent mass versions are described by Gladwell *et al.* (1989).

3.11 Solution of coupled Navier–Stokes problems

For steady state conditions, it was shown in Section 2.16 that a non-linear system of algebraic equations had to be solved, involving, at the element level, submatrices $[\mathbf{c}_{11}]$, $[\mathbf{c}_{12}]$, etc. These element matrices contained velocities $\overline{u}$ and $\overline{v}$ (called `ubar` and `vbar` in the programs) together with shape functions and their derivatives for the velocity and pressure variables. It was mentioned that it would be possible to use different shape functions for the velocity (vector) quantity and pressure (scalar) quantity and this is what is done in programs in Chapters 9 and 12.

The velocity shape functions are designated as `fun` and the pressure shape functions as `funf`. Similarly, the velocity derivatives are `deriv` and the pressure derivatives `derivf`. The arrays `nd1`, `nd2`, `ndf1`, `ndf2`, `nfd1`, and `nfd2` hold the results of cross products between the velocity and pressure shape functions and their derivatives as shown below.

Thus, the element integrals which have to be evaluated numerically from equation (2.115) are of the form

```
dtkd=MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)
CALL cross_product(fun,deriv(1,:),nd1)                    (3.103)
CALL cross_product(fun,deriv(2,:),nd2)
```

followed by

$$\begin{aligned}\texttt{c11 (=c33)} &= \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*dtkd}_i \\ &+ \texttt{ubar}_i \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*nd1}_i \\ &+ \texttt{vbar}_i \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*nd2}_i\end{aligned} \qquad (3.104)$$

In these equations, `deriv(1,:)` signifies the first row of `deriv` and so on in the usual Fortran 95 style. The diagonal terms in `kay` represent the reciprocal of the Reynolds number. Note the identity of the first term of `c11` with (3.63) for uncoupled flow.

The remaining submatrices are formed as follows:

```
CALL cross_product(fun,derivf(1,:),ndf1)
CALL cross_product(fun,derivf(2,:),ndf2)
CALL cross_product(funf,deriv(1,:),nfd1)          (3.105)
CALL cross_product(funf,deriv(2,:),nfd2)
```

followed by

$$\texttt{c12} = \frac{1}{\texttt{rho}} \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*ndf1}_i$$

$$\texttt{c32} = \frac{1}{\texttt{rho}} \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*ndf2}_i$$

$$\texttt{c21} = \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*nfd1}_i \tag{3.106}$$
$$\texttt{c23} = \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*nfd2}_i$$

where `rho` is the mass density. The other submatrices `c13`, `c22`, and `c31` are set to zero.

The (unsymmetrical) matrix built up from these submatrices is called $[\mathbf{k}_e]$, and is formed by a special subroutine called `formupv` (`formupvw` in 3D) and the global, unsymmetrical, band matrix is assembled using `formtb`. The appropriate equation solution routines are `gauss_band` and `solve_band` as shown in Table 3.8. In an element-by-element context, the iterative solution method is BiCGStab(l) following the algorithm described in (3.27) to (3.29).

3.12 Solution of coupled transient problems

The element equations for Biot consolidation were shown in equation (2.139) to be given by

$$\begin{aligned}[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{c}]\{\mathbf{u}_w\} &= \{\mathbf{f}\}\\ [\mathbf{c}]^{\mathrm{T}}\left\{\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right\} - [\mathbf{k}_c]\{\mathbf{u}_w\} &= \{\mathbf{0}\}\end{aligned} \tag{3.107}$$

where $[\mathbf{k}_m]$ and $[\mathbf{k}_c]$ are the now familiar solid stiffness and fluid conductivity matrices. The matrix $[\mathbf{c}]$ is the connectivity matrix which is formed from integrals of the form,

$$\iint \frac{\partial N_j}{\partial x} N_i \,\mathrm{d}x\,\mathrm{d}y \tag{3.108}$$

where the first derivative term comes from the displacement field, and the second term comes from the excess pore pressure field. The programs described in Chapter 9 use different element types for the displacements (8-node) and the excess pore pressures (4-node). In Chapter 12, the 3D elements have 20 displacement and 8 pore pressure nodes.

The integrals that generate the $[\mathbf{c}]$ matrix involve the product of the vectors `vol` and `funf`, where `vol` is derived from the familiar `deriv` array for the solid elements, and takes the form, for 2D,

$$\texttt{vol} = \left[\frac{\partial N_1}{\mathrm{d}x}\ \frac{\partial N_1}{\mathrm{d}y}\ \frac{\partial N_2}{\mathrm{d}x}\ \frac{\partial N_2}{\mathrm{d}y}\cdots\ \cdots\frac{\partial N_8}{\mathrm{d}x}\ \frac{\partial N_8}{\mathrm{d}y}\right]^{\mathrm{T}} \tag{3.109}$$

and `funf` holds the fluid component shape functions as

$$\texttt{funf} = [N_1\ \ N_2\ \ N_3\ \ N_4]^{\mathrm{T}} \tag{3.110}$$

The following sequence completes the integration:

$$\texttt{CALL cross\_product(vol,funf,volf)}$$

$$\texttt{c} = \sum_{i=1}^{\texttt{nip}} \texttt{det}_i\texttt{*weights(i)*volf}_i \tag{3.111}$$

The volumetric strain (ϵ_v) at any point within a displacement element is easily retrieved from the product, `DOT_PRODUCT(vol,eld)` where `eld` holds the element nodal displacements.

To integrate equations (3.107) with respect to time there are again many methods available, but we consider only the simplest linear interpolation in time using finite differences, similar to that used for first order uncoupled problems in equation (3.94).

3.12.1 Absolute load version

This approach applies the full external loading at each time step and is suitable for linear elastic problems. Interpolation in time using θ, and elimination of the derivative terms as was done for first order problems in (3.94), leads to the following recurrence equations at the element level:

$$\begin{bmatrix} \theta\,[\mathbf{k}_m] & \theta\,[\mathbf{c}] \\ \theta\,[\mathbf{c}]^{\mathrm{T}} & -\theta^2\Delta t\,[\mathbf{k}_c] \end{bmatrix} \begin{Bmatrix} \{\mathbf{u}\} \\ \{\mathbf{u}_w\} \end{Bmatrix}_1 = \begin{bmatrix} -(1-\theta)\,[\mathbf{k}_m] & -(1-\theta)\,[\mathbf{c}] \\ \theta\,[\mathbf{c}]^{\mathrm{T}} & \theta(1-\theta)\Delta t\,[\mathbf{k}_c] \end{bmatrix} \begin{Bmatrix} \{\mathbf{u}\} \\ \{\mathbf{u}_w\} \end{Bmatrix}_0 + \begin{Bmatrix} (1-\theta)\{\mathbf{f}\} \\ \{\mathbf{0}\} \end{Bmatrix}_0 + \begin{Bmatrix} \theta\{\mathbf{f}\} \\ \{\mathbf{0}\} \end{Bmatrix}_1 \tag{3.112}$$

where $\{\mathbf{f}\}$ represents the external loading vector which may itself be time dependent.

The left and right hand side element matrices, called $[\mathbf{k}_e]$ and $[\mathbf{k}_d]$ respectively, are formed from their constituent matrices by subroutine `fmkdke`. The second equation has been multiplied through by θ to preserve symmetry of the $[\mathbf{k}_e]$ matrix; however, the right hand side matrix $[\mathbf{k}_d]$ is unsymmetric.

A Crank–Nicolson type of approximation, $\theta = 1/2$ would be a popular choice in equation (3.112); however, it will be shown in Chapter 8 that this approximation can lead to oscillatory results. The oscillations can be smoothed out either by using the fully implicit version with $\theta = 1$, or by writing the first of (3.112) with $\theta = 1$ and the second with $\theta = 1/2$. Examples of this algorithm with constant θ are presented in Chapter 9. In all cases, a right hand side matrix-by-vector multiplication is followed by an equation solution for each timestep. As before a saving in computer time can be achieved if $[\mathbf{k}_m]$, $[\mathbf{c}]$ and $[\mathbf{k}_c]$ are independent of time, and constant Δt is used, because the left hand side matrix needs to be factorised only once.

Note that the left hand side matrix is always symmetrical whereas the right hand side is not. Therefore, if using an assembly approach `fsparv` can be used to assemble the left hand side system equations from (3.112), where `formtb` must be used to assemble the right hand side system followed by `bantmul` to complete the matrix–vector multiply. Element-by-element summation is most effective in the right-hand side operations in (3.112) due to sparsity, and “element-by-element” (mesh-free) algorithms can be developed using pcg equation solution as shown in Chapters 9 and 12 .

3.12.2 Incremental load version

In (3.107), {**f**} is the total force applied and these equations are appropriate to linear systems. Later in this book we shall be concerned with non-linear systems similar to those described in Chapter 6, in which it is desirable to apply loads incrementally and allow plastic stress redistribution to equilibrate at each step.

If $\{\Delta\mathbf{f}\}$ is the change in load between successive times, the incremental form of the first of (3.107) is

$$[\mathbf{k}_m]\{\Delta\mathbf{u}\} + [\mathbf{c}]\{\Delta\mathbf{u}_w\} = \{\Delta\mathbf{f}\} \tag{3.113}$$

where $\{\Delta\mathbf{u}\}$ and $\{\Delta\mathbf{u}_w\}$ are the resulting changes in displacement and excess pore pressure respectively. Linear interpolation in time using the θ-method yields

$$\{\Delta\mathbf{u}\} = \Delta t\left((1-\theta)\left\{\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right\}_0 + \theta\left\{\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right\}_1\right) \tag{3.114}$$

and the second of (3.107) can be written at the two time levels to give expressions for the derivatives which can then be eliminated to give the following incremental recurrence equations (Sandhu and Wilson, 1969; Griffiths, 1994a; Hicks, 1995).

$$\begin{bmatrix} [\mathbf{k}_m] & [\mathbf{c}] \\ [\mathbf{c}]^{\mathrm{T}} & -\theta\Delta t\,[\mathbf{k}_c] \end{bmatrix} \begin{Bmatrix} \{\Delta\mathbf{u}\} \\ \{\Delta\mathbf{u}_w\} \end{Bmatrix} = \begin{Bmatrix} \{\Delta\mathbf{f}\} \\ \Delta t\,[\mathbf{k}_c]\{\mathbf{u}_w\}_0 \end{Bmatrix} \tag{3.115}$$

The left hand side element matrix, again called $[\mathbf{k}_e]$, is formed from its constituent matrices by subroutine `formke` and is symmetric. If using an assembly strategy, subroutine `fsparv` generates the global matrix. The right hand side vector consists of load increments $\{\Delta\mathbf{f}\}$ and fluid “loads” given by $\Delta t\,[\mathbf{k}_c]\{\mathbf{u}_w\}_0$. The fluid term is conveniently computed without any need for assembly using an element-by-element product approach (see Program 9.3). Subroutines `sparin` and `spabac` complete the solution for the incremental displacements and excess pore pressures.

At each time step, all that remains is to update the dependent variables using

$$\begin{aligned} \{\mathbf{u}\}_1 &= \{\mathbf{u}\}_0 + \{\Delta\mathbf{u}\} \\ \{\mathbf{u}_w\}_1 &= \{\mathbf{u}_w\}_0 + \{\Delta\mathbf{u}_w\} \end{aligned} \tag{3.116}$$

3.13 Solution of second order time dependent problems

The basic second-order propagation type of equation was derived in Chapter 2 and at the element level takes the form of (2.102), namely

$$[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{m}_m]\left\{\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}t^2}\right\} = \{\mathbf{f}(t)\} \tag{3.117}$$

where in the context of solid mechanics, $[\mathbf{k}_m]$ is the element elastic stiffness and $[\mathbf{m}_m]$ the element mass. In equation (3.117) a time dependent forcing term $\{\mathbf{f}(t)\}$ has been included on the right hand side. In addition to these elastic and inertial forces, solids in motion experience a third type of force whose action is to dissipate energy. For example, the solid may deform so much that plastic strains result, or may be subjected to internal or external friction. Although these phenomena are non-linear in character and can be treated by the non-linear analysis techniques given in Chapter 6, it has been common to linearise the dissipative forces, for example by assuming that they are proportional to velocity. This allows (3.117) to be modified to

$$[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{c}_m]\left\{\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t}\right\} + [\mathbf{m}_m]\left\{\frac{\mathrm{d}^2\mathbf{u}}{\mathrm{d}t^2}\right\} = \{\mathbf{f}(t)\} \tag{3.118}$$

where $[\mathbf{c}_m]$ is assumed to be a constant element damping matrix.

Although in principle $[\mathbf{c}_m]$ could be independently measured or assessed, it is common practice to assume that $[\mathbf{c}_m]$ is taken to be a linear combination of $[\mathbf{m}_m]$ and $[\mathbf{k}_m]$, where

$$[\mathbf{c}_m] = f_m[\mathbf{m}_m] + f_k[\mathbf{k}_m] \tag{3.119}$$

where f_m and f_k are scalars, the so-called "Rayleigh" damping coefficients. They can be related to the more usual "damping ratio" ζ (Timoshenko *et al.*, 1974) by means of

$$\zeta = \frac{f_m + f_k\omega^2}{2\omega} \tag{3.120}$$

where ω is the natural (usually fundamental) frequency of vibration.

The most generally applicable technique for integrating (3.118) with respect to time is "direct integration" in an analogous way to that previously described for first order problems. Two of the simplest popular implicit methods are described in subsequent sections, where the solution is advanced by one time interval Δt, the values of the displacement and its derivatives at one instant in time being sufficient to determine these values at the subsequent instant by means of recurrence relations. Both preserve unconditional stability, and examples that utilise both element assembly and element-by-element strategies are presented in Chapters 11 and 12.

Attention is first focused however on the "modal superposition" method.

3.13.1 Modal superposition

This method has as its basis the free undamped part of (3.118), that is when $[\mathbf{c}_m]$ and $\{\mathbf{f}\}$ are zero. The reduced equation in assembled form is

$$[\mathbf{K}_m]\{\mathbf{U}\} + [\mathbf{M}_m]\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\} = \{\mathbf{0}\} \tag{3.121}$$

which can of course be converted into an eigenproblem by the assumption of harmonic motion

$$\{\mathbf{U}\} = \{\mathbf{A}\}\sin(\omega t + \psi) \tag{3.122}$$

to give

$$[\mathbf{K}_m]\{\mathbf{A}\} - \omega^2[\mathbf{M}_m]\{\mathbf{A}\} = \{\mathbf{0}\} \tag{3.123}$$

Solution of this eigenproblem by the techniques previously described results in `neq` eigenvalues ω^2 and eigenvectors $\{\mathbf{A}\}$, where `neq` (in program terminology) is the total number of degrees of freedom in the finite element mesh. These eigenvectors or "mode shapes" can be considered to be columns of a modal matrix $[\mathbf{P}]$, where

$$[\mathbf{P}] = \left[\ \{\mathbf{A}_1\}\ \ \{\mathbf{A}_2\}\ \ .\ \ .\ \ \{\mathbf{A}_{nmodes+}\}\ \right] \tag{3.124}$$

where `nmodes` is the number of modes that are contributing to the time response. Often it is not necessary to include the higher frequency components in an analysis, so that `nmodes` $\leq$ `neq`.

Because of the properties of eigenproblems that mode shapes possess orthogonality, one to the other, such that

$$\left.\begin{aligned} \{\mathbf{A}_i\}^{\mathrm{T}}[\mathbf{M}_m]\left\{\mathbf{A}_j\right\} &= 0 \\ \{\mathbf{A}_i\}^{\mathrm{T}}[\mathbf{K}_m]\left\{\mathbf{A}_j\right\} &= 0 \end{aligned}\right\} \quad i \neq j \tag{3.125}$$

$$\left.\begin{aligned} \{\mathbf{A}_i\}^{\mathrm{T}}[\mathbf{M}_m]\left\{\mathbf{A}_j\right\} &= m'_{ii} \\ \{\mathbf{A}_i\}^{\mathrm{T}}[\mathbf{K}_m]\left\{\mathbf{A}_j\right\} &= k'_{ii} \end{aligned}\right\} \quad i = j \tag{3.126}$$

where m'_{ii} and k'_{ii} are the diagonal terms of the diagonal global "principal" mass and stiffness matrices, $[\mathbf{M}']$ and $[\mathbf{K}']$ respectively. Use of these relationships in (3.121) has the effect of uncoupling the equations in terms of the principal or "normal" coordinates $\{\mathbf{U}'\}$, thus

$$[\mathbf{K}']\left\{\mathbf{U}'\right\} + [\mathbf{M}']\left\{\frac{\mathrm{d}^2\mathbf{U}'}{\mathrm{d}t^2}\right\} = \{\mathbf{0}\} \tag{3.127}$$

The effect of uncoupling has been to reduce the vibration problem to a set of `nmodes` independent second order equations (3.127).

The actual displacements can be retrieved from the normal coordinates by a final superposition process given by

$$\{\mathbf{U}\} = [\mathbf{P}]\left\{\mathbf{U}'\right\} \tag{3.128}$$

Inclusion of damping

Free damped vibrations, governed by

$$[\mathbf{K}_m]\{\mathbf{U}\} + [\mathbf{C}_m]\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\} + [\mathbf{M}_m]\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\} = \{\mathbf{0}\} \tag{3.129}$$

can be handled by the above technique if it is assumed that the undamped mode shapes are also orthogonal with respect to the damping matrix $[\mathbf{C}_m]$ in the way described by (3.125) and (3.126). This can readily be achieved if $[\mathbf{C}_m]$ is taken to be a linear combination of $[\mathbf{M}_m]$ and $[\mathbf{K}_m]$,

$$[\mathbf{C}_m] = f_m[\mathbf{M}_m] + f_k[\mathbf{K}_m] \tag{3.130}$$

as defined previously in (3.119). Because of orthogonality with respect to $[\mathbf{C}_m]$, the uncoupled normal coordinate equations are

$$[\mathbf{K}^{'}]\left\{\mathbf{U}^{'}\right\} + [\mathbf{C}^{'}]\left\{\frac{\mathrm{d}\mathbf{U}^{'}}{\mathrm{d}t}\right\} + [\mathbf{M}^{'}]\left\{\frac{\mathrm{d}^2\mathbf{U}^{'}}{\mathrm{d}t^2}\right\} = \{\mathbf{0}\} \tag{3.131}$$

The modal matrix $[\mathbf{P}]$ is usually "mass normalised", leading to

$$\begin{aligned}[\mathbf{M}^{'}] &= [\mathbf{I}]\\ [\mathbf{C}^{'}] &= (f_m + f_k\omega^2)\,[\mathbf{I}]\\ [\mathbf{K}^{'}] &= \omega^2\,[\mathbf{I}]\end{aligned} \tag{3.132}$$

By working in normal coordinates it has become necessary to take a constant ζ for the complete mesh being analysed (see equation 3.120), although ζ could be varied from mode to mode. Since many real systems contain areas with markedly different damping properties, this is an undesirable feature of the method in practice (see Program 11.2).

Inclusion of forcing terms

When $\{\mathbf{f}(t)\}$ is non-zero, the right hand side of a typical modal equation becomes

$$[\mathbf{K}^{'}]\left\{\mathbf{U}^{'}\right\} + [\mathbf{C}^{'}]\left\{\frac{\mathrm{d}\mathbf{U}^{'}}{\mathrm{d}t}\right\} + [\mathbf{M}^{'}]\left\{\frac{\mathrm{d}^2\mathbf{U}^{'}}{\mathrm{d}t^2}\right\} = [\mathbf{P}]^{\mathrm{T}}\{\mathbf{F}(t)\} \tag{3.133}$$

For example, suppose that in a specific problem only degrees of freedom 10 and 12 are loaded with forces $\cos\theta t$. The jth row of (3.133) would be

$$\omega_j^2 U_j^{'} + (f_m + f_k\omega_j^2)\frac{\mathrm{d}U_j^{'}}{\mathrm{d}t} + \frac{\mathrm{d}^2U_j^{'}}{\mathrm{d}t^2} = (P_{10,j} + P_{12,j})\cos\theta t \tag{3.134}$$

or

$$\omega_j^2 U_j^{'} + 2\zeta\omega_j\frac{\mathrm{d}U_j^{'}}{\mathrm{d}t} + \frac{\mathrm{d}^2U_j^{'}}{\mathrm{d}t^2} = P_j^{'}\cos\theta t \tag{3.135}$$

The particular solution to this equation with stationary initial conditions is

$$U_j^{'} = \frac{(\omega_j^2 - \theta^2)P_j^{'}}{(\omega_j^2 - \theta^2)^2 + 4\zeta^2\omega_j^2\theta^2}\cos\theta t + \frac{2\zeta\omega_j\theta P_j^{'}}{(\omega_j^2 - \theta^2)^2 + 4\zeta^2\omega_j^2\theta^2}\sin\theta t \tag{3.136}$$

The normal coordinates having been determined, the actual displacements can be recovered using (3.128).

For more general forcing functions (3.133) must be solved by other means, for example by one of the direct integration methods described below.

3.13.2 Newmark or Crank–Nicolson method

If Rayleigh damping is assumed, a class of recurrence relations based on linear interpolation in time can again be constructed, involving the scalar parameter θ which varies between 1/2 and 1 in the same way as was done for first order problems.

If using an assembly technique, the equations (3.118) are written at both the "0" and "1" time stations,

$$\begin{aligned}
&[\mathbf{K}_m]\{\mathbf{U}\}_0 + (f_m[\mathbf{M}_m] + f_k[\mathbf{K}_m])\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0 + [\mathbf{M}_m]\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0 = \{\mathbf{F}\}_0 \\
&[\mathbf{K}_m]\{\mathbf{U}\}_1 + (f_m[\mathbf{M}_m] + f_k[\mathbf{K}_m])\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_1 + [\mathbf{M}_m]\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_1 = \{\mathbf{F}\}_1
\end{aligned} \tag{3.137}$$

and assuming linear interpolation in time,

$$\begin{aligned}
\{\mathbf{U}\}_1 &= \{\mathbf{U}\}_0 + \Delta t\left((1-\theta)\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0 + \theta\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_1\right) \\
\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_1 &= \left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0 + \Delta t\left((1-\theta)\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0 + \theta\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_1\right)
\end{aligned} \tag{3.138}$$

Rearrangement of these equations and elimination of acceleration terms leads to the following three recurrence relations,

$$\begin{aligned}
&\left[\left(f_m + \frac{1}{\theta\Delta t}\right)[\mathbf{M}_m] + (f_k + \theta\Delta t)[\mathbf{K}_m]\right]\{\mathbf{U}\}_1 \\
&\quad = \theta\Delta t\{\mathbf{F}\}_1 + (1-\theta)\Delta t\{\mathbf{F}\}_0 + \left(f_m + \frac{1}{\theta\Delta t}\right)[\mathbf{M}_m]\{\mathbf{U}\}_0 \\
&\quad + \frac{1}{\theta}[\mathbf{M}_m]\left\{\frac{d\mathbf{U}}{dt}\right\}_0 + (f_k - (1-\theta)\Delta t)[\mathbf{K}_m]\{\mathbf{U}\}_0
\end{aligned} \tag{3.139}$$

$$\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_1 = \frac{1}{\theta\Delta t}(\{\mathbf{U}\}_1 - \{\mathbf{U}\}_0) - \frac{1-\theta}{\theta}\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0 \tag{3.140}$$

$$\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_1 = \frac{1}{\theta\Delta t}\left(\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_1 - \left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0\right) - \frac{1-\theta}{\theta}\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0 \tag{3.141}$$

The algorithm requires initial conditions on displacements $\{\mathbf{U}\}_0$ and velocities $\{\mathrm{d}\mathbf{U}/\mathrm{d}t\}_0$ to be provided in order to get started.

In the special case when $\theta = 1/2$ this method is Newmark's "$\beta = 1/4$" method, which is also the exact equivalent of the Crank–Nicolson method used in first order problems. There are other variants of the Newmark type, but this is the most common.

The principal recurrence relation (3.139) is clearly similar to those which arose in first order problems, for example (3.94). Although substantially more matrix-by-vector multiplications are involved on the right hand side, together with matrix and vector additions, the recurrence again consists essentially of an equation solution per time step. Advantage can as usual be taken of a constant left-hand side matrix should this occur, and element-by-element strategies are easily implemented via pcg (see Chapters 11 and 12).

3.13.3 Wilson's method

Assuming again an assembly approach, the equations (3.118) are advanced from some known state $\{\mathbf{U}\}_0$, $\{\mathrm{d}\mathbf{U}/\mathrm{d}t\}_0$, and $\left\{\mathrm{d}^2\mathbf{U}/\mathrm{d}t^2\right\}_0$ to the new solution $\{\mathbf{U}\}_1$, $\{\mathrm{d}\mathbf{U}/\mathrm{d}t\}_1$, and $\left\{\mathrm{d}^2\mathbf{U}/\mathrm{d}t^2\right\}_1$ an interval Δt later by first linearly extrapolating to a hypothetical solution, say $\{\mathbf{U}\}_2$, $\{\mathrm{d}\mathbf{U}/\mathrm{d}t\}_2$, and $\left\{\mathrm{d}^2\mathbf{U}/\mathrm{d}t^2\right\}_2$ an interval $\delta t = \theta\Delta t$ later where $1.4 \le \theta \le 2$.

If Rayleigh damping is again assumed, $\{\mathbf{U}\}_2$ is first computed from

$$\left[\left(\frac{6}{\theta^2\Delta t^2}+\frac{3f_m}{\theta\Delta t}\right)[\mathbf{M}_m]+\left(\frac{3f_k}{\theta\Delta t}+1\right)[\mathbf{K}_m]\right]\{\mathbf{U}\}_2=\{\mathbf{F}\}_2+\left[\left(\frac{6}{\theta^2\Delta t^2}+\frac{3f_m}{\theta\Delta t}\right)\{\mathbf{U}\}_0\right.$$
$$\left.+\left(\frac{6}{\theta\Delta t}+2f_m\right)\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0+\left(2+\frac{f_m\theta\Delta t}{2}\right)\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0\right][\mathbf{M}_m]$$
$$+\left[\frac{3f_k}{\theta\Delta t}\{\mathbf{U}\}_0+2f_k\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0+\frac{f_k\theta\Delta t}{2}\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0\right][\mathbf{K}_m] \tag{3.142}$$

where

$$\{\mathbf{F}\}_2=(1-\theta)\{\mathbf{F}\}_0+\theta\{\mathbf{F}\}_1 \tag{3.143}$$

The acceleration at the hypothetical station can then be computed from

$$\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_2=\frac{6}{\theta^2\Delta t^2}(\{\mathbf{U}\}_2-\{\mathbf{U}\}_0)-\frac{6}{\theta\Delta t}\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0-2\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0 \tag{3.144}$$

and thus the acceleration at the true station can be interpolated or "averaged" using

$$\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_1=\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0+\frac{1}{\theta}\left(\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_2-\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0\right) \tag{3.145}$$

A Crank–Nicolson equation then gives the desired velocity from

$$\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_1=\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0+\frac{\Delta t}{2}\left(\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0+\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_1\right) \tag{3.146}$$

and the updated displacements from

$$\{\mathbf{U}\}_1=\{\mathbf{U}\}_0+\Delta t\left\{\frac{\mathrm{d}\mathbf{U}}{\mathrm{d}t}\right\}_0+\frac{\Delta t^2}{3}\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_0+\frac{\Delta t^2}{6}\left\{\frac{\mathrm{d}^2\mathbf{U}}{\mathrm{d}t^2}\right\}_1 \tag{3.147}$$

The principal recurrence relation (3.142) is again of the familiar type for all one-step time integration methods.

3.13.4 Explicit methods and other storage-saving strategies

The implicit methods described above are relatively safe to use due to their unconditional stability. However, as was the case for first order problems, storage demands become considerable for large systems, and so can solution times for non-linear problems (even though refactorisation of the left hand side of (3.139) or (3.142) is not usually necessary, the non-linear effects having been transposed to the right hand side).

Using a pcg strategy, the implicit equation solution can always be done element-by-element, but the simplest option is the analogue of (3.98), in which θ is set to zero and the mass matrix lumped. In the resulting explicit algorithm, operations are carried out element-wise and no global system storage is necessary (see e.g. Program 11.7). Of course the drawback is potential loss of stability, so that stable time steps may need to be very small indeed.

Since stability is governed by the highest natural frequency of the numerical approximation and since such high frequencies are derived from the stiffest elements in the system, it is quite possible to implement hybrid methods in which the very stiff elements are integrated implicitly, but the remainder are integrated explicitly. Equation solution as implied by (3.139), for example, is still necessary, but the half-bandwidth of the rows in the coefficient matrix associated with freedoms in explicit elements not connected to implicit ones is only one. Thus great savings in storage can be made (Smith, 1984).

Another alternative is to resort to operator splitting, as was done in first order problems. In Chapter 11, implicit, explicit and mixed implicit/explicit algorithms are described and listed, with alternative assembly or EBE solution for the implicit cases. Although product EBE methods have been developed (Wong *et al.*, 1989) they are beyond the scope of the present book.

References

Bai Z, Demmel J, Dongarra J, Ruhe A and der Vorst HV 2000 *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM Press, Philadelphia, Pa.

Bathe KJ 1996 *Numerical Methods in Finite Element Analysis*, 3rd edn. Prentice Hall, Englewood Cliffs, N.J.

Cardoso JP 1994 *Generation of Finite Element Matrices Using Computer Algebra*. Masters thesis, School of Engineering, University of Manchester.

Chan SH, Phoon KK and Lee FH 2001 A modified Jacobi preconditioner for solving ill-conditioned Biot's consolidation equations using symmetric quasi-minimal residual method. *Int J Numer Anal Methods Geomech* **25**(10), 1001–1025.

Ergatoudis J, Irons BM and Zienkiewicz OC 1968 Curved isoparametric quadrilateral elements for finite element analysis. *Int J Solids Struct* **4**, 31.

Gladwell I, Smith IM, Gilvary B and Wong SW 1989 A consistent mass EBE algorithm for linear parabolic systems. *Commun Appl Numer Methods* **5**, 229–235.

Greenbaum A 1997 *Iterative Methods for Solving Linear Systems*. SIAM Press, Philadelphia, Pa.

Griffiths DV 1991 Generalised numerical integration of moments. *Int J Numer Methods Eng* **32**(1), 129–147.

Griffiths DV 1994a Coupled analyses in geomechanics. In *Visco-Plastic Behavior of Geomaterials* (eds. Cristescu ND and Gioda G). Springer-Verlag, Wien, New York pp. 245–317. Chapter 5.

Griffiths DV 1994b Stiffness matrix of the 4-node quadrilateral element in closed-form. *Int J Numer Methods Eng* **37**(6), 1027–1038.

Griffiths DV 2004 Use of computer algebra systems in finite element software development. *Proceedings of the 7th International Congress on Numerical Methods in Engineering and Scientific Applications*, CIMENICS '04 (ed. Rojo J *et al.*) Sociedad Venezolana de Métodos Numéricos en Ingenierıa, Caracas, Venezuela, pp. CI 55–66.

Griffiths DV and Smith IM 1991 *Numerical Methods for Engineers*. Blackwell Scientific Publications Ltd., Oxford.

Hicks MA 1995 MONICA-a computer algorithm for solving boundary value problems using the double hardening constitutive model Monot: I algorithm development. *Int J Numer Anal Methods Geomech* **19**, 11–27.

HSL 2002 *A Collection of Fortran Codes for Large Scientific Computation*. See `http://www.cse.clrc.ac.uk/nag/hsl/`.

Hughes TJR, Levit I and Winget J 1983 Element by element implicit algorithms for heat conduction. *J Eng Mech, ASCE* **109**(2), 576–585.

Irons BM 1966a Numerical integration applied to finite element methods. *Conference on the Use of Digital Computers in Structural Engineering*, University of New Castle, New Castle, Pa.

Irons BM 1966b Engineering applications of numerical integration in stiffness method. *J Am Inst Aeronaut Astronaut* **14**, 2035.

Irons BM 1971 Quadrature rules for brick-based finite elements. *Int J Numer Meths Eng* **3**, 293-294.

Jennings A and McKeown JJ 1992 *Matrix Computation*. John Wiley & Sons, Chichester, New York.

Kelley CT 1995 *Iterative Methods for Linear and Nonlinear Equations*. SIAM Press, Philadelphia, Pa.

Kopal A 1961 *Numerical Analysis*, 2nd edn. Chapman & Hall, London, New York.

Parlett BN and Reid JK 1981 Tracking the progress of the Lanczos algorithm for large symmetric eigenproblems. *IMA J Numer Anal* **1**, 135–155.

Sandhu RS and Wilson EL 1969 Finite-element analysis of seepage in elastic media. *J Eng Mech, ASCE* **95**(EM3), 641–652.

Sleijpen GLG and van der Vorst HA 2000 Jacobi-Davidson methods. In *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide* (eds. Bai Z. Demmel, J. Dongarra, J. Ruhe, A. and van der Vorst, H. SIAM Press Philadelphia, Pa.

Sleijpen GLG, van der Vorst HA and Fokkema DR 1994 BiCGStab(l) and other hybrid Bi-CG methods. *Numer Algorithms* **7**, 75–109.

Smith IM 1979 Discrete element analysis of pile instability. *Int J Numer Anal Methods Geomech* **3**, 205–211.

Smith IM 1984 Adaptability of truly modular software. *Eng Comput* **1**(1), 25–35.

Smith IM 2000 A general purpose system for finite element analyses in parallel. *Eng Comput* **17**(1), 75–91.

Smith IM and Kidger DJ 1991 Properties of the 20-node brick element. *Int J Numer Anal Methods Geomech* **15**(12), 871–891.

Smith IM and Kidger DJ 1992 Elastoplastic analysis using the 14-node brick element family. *Int J Numer Methods Eng* **35**, 1263–1275.

Smith IM, Wong SW, Gladwell I and Gilvary B 1989 PCG methods in transient FE analysis Part I: first order problems. *Int J Numer Methods Eng* **28**(7), 1557–1566.

Taig IC 1961 Structural analysis by the matrix displacement method. Technical Report SO17, English Electric Aviation Report, Preston.

Timoshenko SP, Young D and Weaver W 1974 *Vibration Problems in Engineering*, 4th edn. John Wiley & Sons, Chichester, New York.

Wong SW, Smith IM and Gladwell I 1989 PCG methods in transient FE analysis Part II: second order problems. *Int J Numer Methods Eng* **28**(7), 1567–1576.

Zienkiewicz OC, Too J and Taylor RL 1971 Reduced integration technique in general analysis of plates and shells. *Int J Numer Methods Eng* **3**, 275–290.

Zienkiewicz OC, Irons BM, Ergatoudis J, Ahmad S and Scott FC 1969 Isoparametric and associated element families for two and three dimensional analysis. In *Proceedings of a Course on Finite Element Methods in Stress Analysis* (eds. Holland I and Bell K). Norwegian University of Science and Technology, Trondheim, Norway

4

Static Equilibrium of Structures

4.1 Introduction

Practical finite element analysis had as its starting point matrix analysis of "structures", by which engineers usually mean assemblages of elastic, line elements. The matrix displacement (stiffness) method is a special case of finite element analysis, and since many engineers still begin their acquaintance with the finite element method in this way, the opening applications chapter of this book is devoted to "structural" analysis.

The first program, Program 4.1, permits the analysis of a rod subjected to combinations of axial loads and displacements at various points along its length. Each 1D rod element can have a different length and axial stiffness but the element stiffness matrices, being simple functions of these two quantities are easily formed by a subroutine. Indeed, in nearly all the programs in this chapter, the element stiffness matrices consist of simple explicit expressions which are conveniently provided by subroutines. Program 4.2 introduces a more general treatment involving rod elements, allowing analyses to be performed of 2D or 3D pin-jointed frames.

Program 4.3 permits the analysis of slender beams subjected to combinations of transverse and moment loading. Optionally, the program allows the inclusion of an elastic foundation enabling analysis of problems generally classified as "beams on elastic foundations".

When 1D beam and rod elements are superposed, the result is a "beam–rod" element, which is a powerful general element that can sustain axial, transverse, and moment loading. This element is able to analyse all conventional structural frames. Program 4.4 implements the "beam–rod" elements in the analysis of two- or three-dimensional framed structures.

Program 4.5 introduces material non-linearity in the form of an elastic-perfectly plastic moment/curvature relationship for beams. The program can compute plastic collapse of one-, two-, or three-dimensional structures when subjected to incrementally changing loads. The non-linearity is dealt with using an iterative, constant stiffness (modified Newton–Raphson) approach. Unlike more traditional approaches, at each iteration the internal loads on the structure are altered rather than the stiffness matrix itself. This approach will be used extensively in the elastic–plastic analyses of solids described in later chapters of the book, notably Chapter 6.

Program 4.6 performs elastic stability analysis of axially loaded beams. As in Program 4.3, an elastic foundation is optional. The program computes the lowest buckling load by iteratively obtaining the smallest eigenvalue of the system based on its stiffness and geometric properties.

The final program in the chapter, Program 4.7, describes a method for analysing rectangular thin plates in bending. This could be considered to be the first "genuine" finite element in the book. The plate stiffness matrix is formed using numerical integration, anticipating the solid mechanics applications of Chapter 5 and beyond.

In the interests of generality and portability, all programs in the book assume that the data and results file have the generic names `fe95.dat` and `fe95.res`, respectively. When running the programs, it is suggested that a small batch file or shell script is developed by the user that copies the actual data file name to `fe95.dat` before execution. Similarly, after the program has run, the generic results file `fe95.res` should be copied or moved to the results file name required.

Program 4.1 One-dimensional analysis of axially loaded elastic rods using 2-node rod elements.

```
PROGRAM p41
!-------------------------------------------------------------------------
! Program 4.1 One dimensional analysis of axially loaded elastic rods
!             using 2-node rod elements.
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndof=2,nels,neq,nod=2,      &
   nodof=1,nn,nprops=1,np_types,nr
 REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),kdiag(:),nf(:,:),no(:),      &
   node(:),num(:)
 REAL(iwp),ALLOCATABLE::action(:),eld(:),ell(:),km(:,:),kv(:),loads(:),   &
   prop(:,:),value(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,np_types; nn=nels+1
 ALLOCATE(g(ndof),num(nod),nf(nodof,nn),etype(nels),ell(nels),eld(ndof),  &
   km(ndof,ndof),action(ndof),g_g(ndof,nels),prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g;
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------global stiffness matrix assembly------------------
 kv=zero
 elements_2: DO iel=1,nels
```

```
   CALL rod_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!-----------------------read loads and/or displacements-------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),no(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(1,node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
!-----------------------equation solution -------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 WRITE(11,'(/A)')"  Node   Disp"
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(nf(:,k)); END DO
!-----------------------retrieve element end actions----------------------
 WRITE(11,'(/A)')" Element Actions"
 elements_3: DO iel=1,nels
   CALL rod_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
   eld=loads(g); action=MATMUL(km,eld); WRITE(11,'(I5,2E12.4)')iel,action
 END DO elements_3
STOP
END PROGRAM p41
```

Scalar integers:

`fixed_freedoms`	number of fixed displacements
`i`	simple counter
`iel`	simple counter
`iwp`	`SELECTED_REAL_KIND(15)`
`k`	simple counter
`loaded_nodes`	number of loaded nodes
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nn`	number of nodes in the mesh
`nod`	number of nodes per element
`nodof`	number of degrees of freedom per node
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes

Scalar reals:

`penalty`	set to 1×10^{20}
`zero`	set to 0.0

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`kdiag`	diagonal term location vector

`nf`	nodal freedom matrix
`no`	fixed freedoms vector
`node`	fixed nodes vector
`num`	element node numbers vector

Dynamic real arrays:

`action`	element nodal action vector
`eld`	element displacement vector
`ell`	element lengths vector
`km`	element stiffness matrix
`kv`	global stiffness matrix
`loads`	global load (displacement) vector
`prop`	element properties matrix
`value`	fixed displacements vector

The main features of this program are the elastic rod element stiffness matrix (2.11) and the global stiffness matrix assembly described in Section 3.7. The structure chart in Figure 4.1 gives the main sequence of operations.

Program 4.1 is illustrated by two examples shown in Figures 4.2 and 4.3. In both cases, the rod is restrained at one end and free at the other. The rod in Figure 4.2 is subjected to a uniformly distributed axial force of 5.0/unit length, and the rod in Figure 4.3 is subjected to a fixed displacement of 0.05 at its tip.

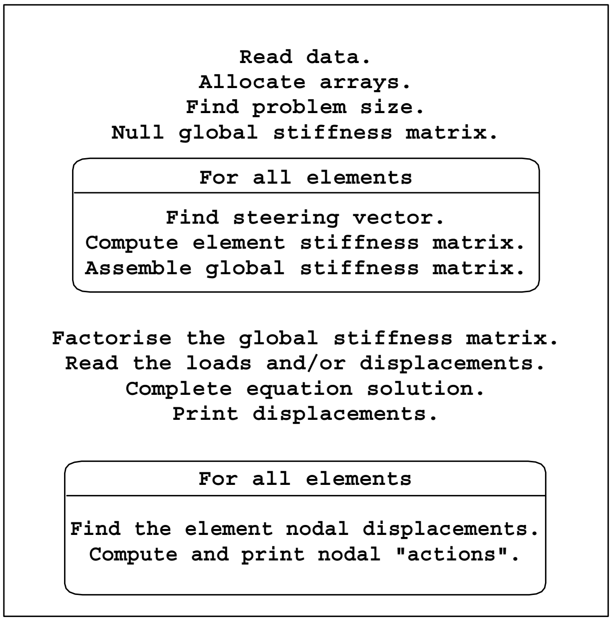
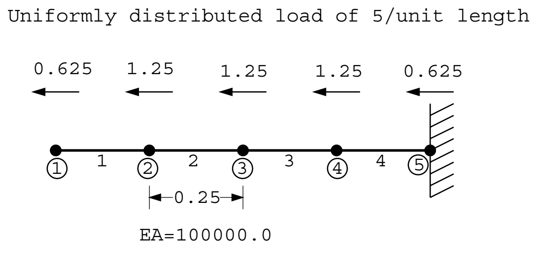
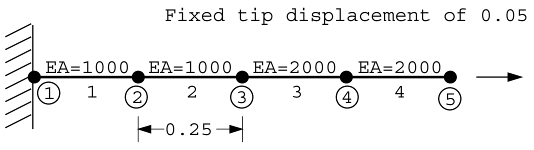

Read data.
Allocate arrays.
Find problem size.
Null global stiffness matrix.
For all elements
Find steering vector.
Compute element stiffness matrix.
Assemble global stiffness matrix.
Factorise the global stiffness matrix.
Read the loads and/or displacements.
Complete equation solution.
Print displacements.
For all elements
Find the element nodal displacements.
Compute and print nodal "actions".

Figure 4.1 Structure chart for Program 4.1



```
nels    np_types
4         1

prop (ea)
100000.0

etype(not needed)

ell
0.25  0.25  0.25  0.25

nr,(k,nf(:,k),i=1,nr)
1
5 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
5
1 -0.625  2 -1.25  3 -1.25  4 -1.25  5 -0.625

fixed_freedoms
0
```

Figure 4.2 Mesh and data for first Program 4.1 example



```
nels    np_types
4         2

prop (ea)
2000.0  1000.0

etype
2  2  1  1

ell
0.25  0.25  0.25  0.25

nr,(k,nf(:,k),i=1,nr)
1
1 0

loaded_nodes
0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
1
5  0.05
```

Figure 4.3 Mesh and data for second Program 4.1 example

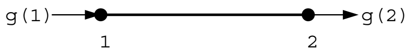


Figure 4.4 Node and freedom numbering for rod elements

Each node has one degree of freedom, namely the axial displacement. The global node numbering system reads from the left and, at the element level, node one is always to the left and node two to the right as shown in Figure 4.4. As explained in Chapter 3, the nodal freedom numbering associated with each element, accounting for any restraints, is contained in the "steering" vector g. Thus, with reference to Figures 4.3 and 4.4, g for element one would be $[0\ 1]^T$ and for element two, $[1\ 2]^T$, and so on.

If a zero appears in the "steering" vector g, this means that the corresponding displacement is fully fixed and equal to zero, as at the right end of the example in Figure 4.2, and the left end of the example in Figure 4.3. In such cases, the "zero freedom" is not assembled into the global matrices.

Nodal freedom data concerning boundary restraints is read by the main program, and takes the form of the number of restrained nodes, nr, followed by, for each restrained node, the restrained node number and a zero. The default is that nodes are not restrained. For problems such as this where 1D elements are strung together in a line, it is a simple matter to automate the generation of the g vector for each element. This is done by the library subroutine num_to_g which picks the correct entries out of the nodal freedom array nf generated by subroutine formnf (see Appendix D for listings of all special purpose subroutines).

Many programs in the book use this approach for defining boundary conditions. In cases later on, where nodes have more than one degree of freedom, some of those nodal freedoms may be restrained and others not. In these cases, the convention will then be to give the node number followed by either ones or zeros (in the correct sequence), where the latter implies a restrained (zero) degree of freedom and the former an unrestrained freedom.

Returning to the main program, some scalar quantities are defined by the type of element being used. For example there can only be 2 nodes per element, so nod=2 and these are assigned in the declaration lines.

The "input and initialisation" section reads the number of elements nels and the number of property types np_types. If there is only one property type (np_types=1) as in the example shown in Figure 4.2, then the property is read and automatically allocated to all elements. If there is more than one property type, as in the example shown in Figure 4.3, then the properties are read, followed by the reading of an integer vector etype which holds information on which property is assigned to which element. The length of each element is read into the real vector ell, and the boundary condition data is read enabling the formation of array nf.

Inside the "global stiffness matrix assembly" section, the element stiffnesses and lengths are used by subroutine rod_km to compute the element stiffness matrices km.

The global stiffness matrix (stored as a skyline column vector kv, see Section 3.7.10) is then assembled for all elements in turn by the library subroutine fsparv once the "steering" vector g has been retrieved. Gaussian elimination on the system equations is split into a "factorisation" phase performed by library subroutine sparin and a forward- and back-substitution phase performed by library subroutine spabac. The "loading" data is then read, and this takes the form of information relating to nodal forces and/or fixed nodal displacements.

In the case of loads, `loaded_nodes` is read first signifying the number of nodes with forces applied. Then for each of these, the node number and the applied force are read. In this rod example there is only one freedom at each node, but in later programs in the chapter where more than one freedom exists at each node, all the "forces" applied at the loaded node must be included in the correct sense (even if some of them are zero).

In the case of fixed displacements, `fixed_freedoms` is read, signifying the number of fixed freedoms in the mesh. Then for each of these, the node number and the value to which the freedom is to be fixed is read. In later programs in the chapter where more than one freedom exists at each node, data must also be read into the vector `sense`, which gives the sequential number of the freedom at the node that is to be fixed. If a particular node has more than one fixed freedom, the node must be entered in the data list for each fixed "sense". If either `loaded_nodes` or `fixed_freedoms` equals zero, no further data relating to that category is required.

In the case shown in Figure 4.2, a rod of uniform stiffness equal to 10^5 is subjected to a (negative) uniformly distributed axial load of 5/unit length. The force has been "lumped" at the nodes as was indicated in (2.10). In this example, there are no fixed displacements, so `fixed_freedoms` is read as zero.

In the case shown in Figure 4.3, the rod has a non-uniform stiffness, and since there is more than one property group (`np_types = 2`), the element type vector `etype` must be read indicating in this case that elements 1 and 2 have an axial stiffness of 1000.0, and elements 3 and 4 have an axial stiffness of 2000.0. There are no loaded nodes so `loaded_nodes` is read as zero but a fixed displacement is applied to the tip of the rod, so `fixed_freedoms` is read as 1, followed by the node number (5) and the magnitude of the fixed displacement (0.05).

Following equation solution, the nodal displacements (overwritten as `loads`) are computed and printed. A final "post-processing" phase is then performed, in which the elements are scanned once more. In this loop, the element nodal displacements (`eld`) are retrieved from the global displacements vector and the element stiffness matrices (`km`) re-computed. Multiplication of the element nodal displacements by the element stiffness matrix using `MATMUL`, results in the element "actions" vector called `action`, which holds the internal end reaction forces for each element.

The computed results for both cases are reproduced in Figure 4.5. In the first case, the end deflection at node 1 is given as -0.25×10^{-4} which is the exact solution. Note, however, that the element "actions" indicate that the fourth element sustains a mean tensile force of 4.375 which is the best this element can do to approximate the true solution of a linearly varying axial load.

In the second case, the nodal displacements indicate the distribution of displacements along the length of the rod up to the end node 5, which has the expected displacement of 0.05. All the elements in the rod are sustaining a tensile load of 66.67.

In problems such as this, the nodal displacements are always in exact agreement with the closed form solution achieved by direct integration of the governing equation. Between the nodes however, the solution may be approximate, due to the limitations of the shape functions.

```
There are    4 equations and the skyline storage is    7

 Node   Disp
    1 -0.2500E-04
    2 -0.2344E-04
    3 -0.1875E-04
    4 -0.1094E-04
    5  0.0000E+00

Element Actions
    1 -0.6250E+00  0.6250E+00
    2 -0.1875E+01  0.1875E+01
    3 -0.3125E+01  0.3125E+01
    4 -0.4375E+01  0.4375E+01


There are    4 equations and the skyline storage is    7

 Node   Disp
    1  0.0000E+00
    2  0.1667E-01
    3  0.3333E-01
    4  0.4167E-01
    5  0.5000E-01

Element Actions
    1 -0.6667E+02  0.6667E+02
    2 -0.6667E+02  0.6667E+02
    3 -0.6667E+02  0.6667E+02
    4 -0.6667E+02  0.6667E+02
```

Figure 4.5 Results from Program 4.1 examples

Program 4.2 Analysis of elastic pin-jointed frames using 2-node rod elements in two or three dimensions.

```
PROGRAM p42
!-------------------------------------------------------------------------
! Program 4.2 Analysis of elastic pin-jointed frames using 2-node rod
!             elements in 2- or 3-dimensions
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim,ndof=2,nels,neq,nod=2, &
   nodof,nn,nprops=1,np_types,nr
 REAL(iwp)::axial,penalty=1.0e20_iwp,zero=0.0_iwp
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::action(:),coord(:,:),eld(:),g_coord(:,:),km(:,:), &
   kv(:),loads(:),prop(:,:),value(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,nn,ndim,np_types; nodof=ndim; ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),km(ndof,ndof),coord(nod,ndim),g_coord(ndim,nn),    &
   eld(ndof),action(ndof),g_num(nod,nels),num(nod),g(ndof),g_g(ndof,nels),&
   etype(nels),prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)g_coord; READ(10,*)g_num
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----------------------loop the elements to find global array sizes-------
 elements_1: DO iel=1,nels
   num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!------------------------global stiffness matrix assembly------------------
 kv=zero
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   CALL pin_jointed(km,prop(1,etype(iel)),coord); g=g_g(:,iel)
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!------------------------read loads and/or displacements-------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),no(fixed_freedoms),                      &
     sense(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
!------------------------equation solution --------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 WRITE(11,'(/A)')  "  Node   Displacement(s)"
 DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!------------------------retrieve element end actions----------------------
 WRITE(11,'(/A)')" Element Actions"
 elements_3: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   CALL pin_jointed(km,prop(1,etype(iel)),coord); g=g_g(:,iel)
   eld=loads(g); action=MATMUL(km,eld); WRITE(11,'(I5,6E12.4)')iel,action
   CALL glob_to_axial(axial,action,coord)
   WRITE(11,'(A,E12.4)')"       Axial force =",axial
 END DO elements_3
STOP
END PROGRAM p42
```

New scalar integers:

`ndim`	number of dimensions

New scalar reals:

`axial`	element axial force

New dynamic integer arrays:

`g_num`	global element node numbers matrix
`sense`	sense of freedoms to be fixed vector

New dynamic real arrays:

`coord`	element nodal coordinates
`g_coord`	nodal coordinates for all elements

The philosophy used throughout this book is to explore solutions to new problems by making gradual alterations to previously described programs, hence this section begins with a listing of variables that have not been used so far in this chapter. This program therefore, is an adaptation of the previous one to allow analysis of rod elements in two or three dimensions. Since rod elements can only sustain axial loads, the types of structural systems for which this is applicable are pin-jointed frames (in 2D) or space structures (in 3D).

The previous program considered rod elements in 1D joined end-to-end, in which the number of nodes was always one greater than the number of elements. This will no longer be true for general two- or three-dimensional structures, so the number of nodes `nn` is now included as data together with the dimensionality of the problem `ndim`.

The variable names are virtually the same as in the previous program. The real array `ell` has been discarded because the rod element lengths are more conveniently calculated from their nodal coordinates. The variable `axial` has been introduced to hold the axial force sustained by each member, and the integer vector `sense` holds the sense of any nodal fixed freedoms, since there is now more than one freedom at each node.

Nodal coordinates must be provided as data and read directly into array `g_coord`, followed by data relating to the element node numbers, read into `g_num`. Two new library subroutines are also introduced. Subroutine `pin_jointed` computes the element stiffness matrix `km`, and subroutine `glob_to_loc` transforms the element "actions" into the axial force held in `axial`.

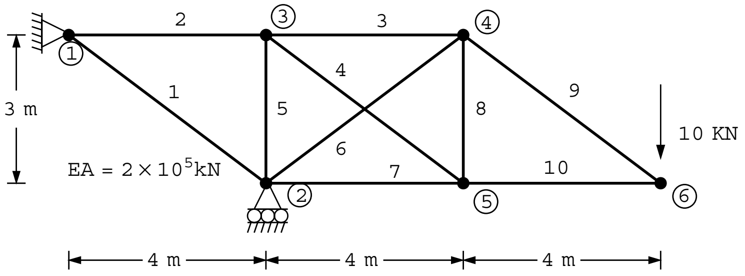
The first example to be solved by Program 4.2 is the 2D pin-jointed frame shown in Figure 4.6. Each element now has 4 degrees of freedom with an x- and a y-translation permitted at each node as shown in Figure 4.7.

In a small problem such as this, the order in which the nodes are numbered is immaterial. In larger problems however, nodal numbering should be made in the most economical order in order to minimise the skyline bandwidths and hence the storage requirements (see Section 3.7.10). In very large problems which use an assembly strategy, a bandwidth optimiser would be used. The loaded nodes and fixed freedoms data follow the same procedure as described in the previous program. In this example, a single load of -10.0 is applied in the y-direction at node 6. Note how the load in the x-direction at this node has also to be read in as zero. There are no fixed freedoms in this example, hence `fixed_freedoms` is read as zero, indicating no further data is needed.

The results given in Figure 4.8 indicate the nodal displacements, followed by the end "actions" and axial force in each element. The results indicate that the displacement under the load is -0.007263 and the axial load in element number 1 is -33.33 (compressive).

The second example to be solved by Program 4.2 is the 3D pin-jointed frame shown in Figure 4.9. Each element now has 6 degrees of freedom with an x-, y-, and z-translation permitted at each node as shown in Figure 4.10.

The data organisation is virtually the same as in the previous example. The dimensionality is increased to `ndim=3` and there are correspondingly three coordinates at each node and three freedoms to be defined at each restrained node. The space–frame shown in Figure 4.9 represents a pyramid-like structure loaded by a force at its apex with components in the x-,



```
nels  nn  ndim   np_types
10     6    2        1

prop(ea)
2.0e5

etype(not needed)

g_coord
0.0 3.0   4.0 0.0   4.0 3.0
8.0 3.0   8.0 0.0  12.0 0.0

g_num
1 2  1 3  3 4  3 5  3 2
2 4  2 5  5 4  4 6  5 6

nr,(k,nf(:,k),i=1,nr)
2
1 0 0  2 1 0

loaded_nodes,(k,loads(nf(:,k))
1
6  0.0 -10.0

fixed_freedoms
0
```

Figure 4.6 Mesh and data for first Program 4.2 example

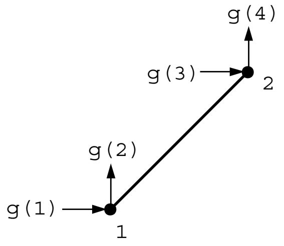
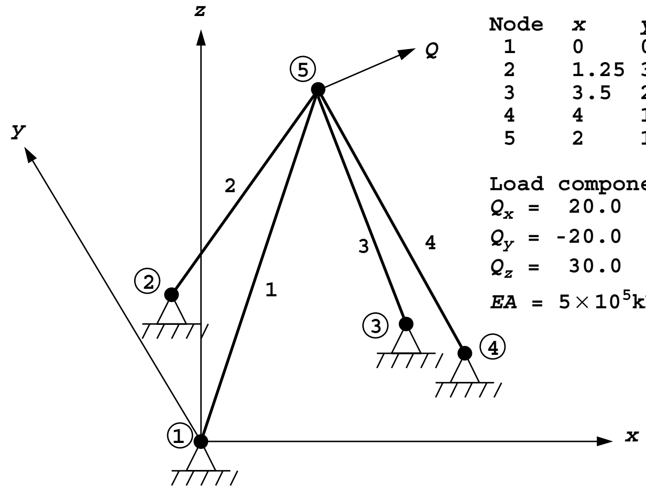


Figure 4.7 Node and freedom numbering for 2D rod elements

y-, and z-directions of 20, −20, and 30 respectively. In addition, the y-freedom (sense 2) at the apex is displaced by −0.0005. The computed results shown in Figure 4.11 indicate that the corresponding displacement components of the loaded node are 0.2569×10^{-3}, -0.5×10^{-3} (as would be expected), and 0.7614×10^{-4}. The axial force in element number 2 is computed to be 49.57 (tensile). It may be noted that in cases such as this where

```
There are    9 equations and the skyline storage is   39

 Node   Displacement(s)
   1  0.0000E+00  0.0000E+00
   2 -0.1042E-02  0.0000E+00
   3  0.5333E-03 -0.6562E-04
   4  0.9500E-03 -0.3046E-02
   5 -0.1425E-02 -0.2981E-02
   6 -0.1692E-02 -0.7263E-02

Element Actions
   1  0.2667E+02 -0.2000E+02 -0.2667E+02  0.2000E+02
      Axial force = -0.3333E+02
   2 -0.2667E+02  0.0000E+00  0.2667E+02  0.0000E+00
      Axial force =  0.2667E+02
   3 -0.2083E+02  0.0000E+00  0.2083E+02  0.0000E+00
      Axial force =  0.2083E+02
   4 -0.5833E+01  0.4375E+01  0.5833E+01 -0.4375E+01
      Axial force =  0.7292E+01
   5  0.0000E+00 -0.4375E+01  0.0000E+00  0.4375E+01
      Axial force = -0.4375E+01
   6  0.7500E+01  0.5625E+01 -0.7500E+01 -0.5625E+01
      Axial force = -0.9375E+01
   7  0.1917E+02  0.0000E+00 -0.1917E+02  0.0000E+00
      Axial force = -0.1917E+02
   8  0.0000E+00  0.4375E+01  0.0000E+00 -0.4375E+01
      Axial force = -0.4375E+01
   9 -0.1333E+02  0.1000E+02  0.1333E+02 -0.1000E+02
      Axial force =  0.1667E+02
  10  0.1333E+02  0.0000E+00 -0.1333E+02  0.0000E+00
      Axial force = -0.1333E+02
```

Figure 4.8 Results from first Program 4.2 example



Nodal coords (m)

Node	x	y	z
1	0	0	0
2	1.25	3	0
3	3.5	2	0
4	4	1	0
5	2	1.5	3

Load components (kN)

Q_x = 20.0

Q_y = -20.0

Q_z = 30.0

$EA = 5 \times 10^5$kN

Figure 4.9 Mesh and data for second Program 4.2 example (*Continued on page 121*)

```
nels  nn  ndim   np_types
4      5    3        1

prop(ea)
5.0e5

etype(not needed)

g_coord
0.0  0.0  0.0    1.25  3.0   0.0    3.5   2.0   0.0
4.0  1.0  0.0    2.0   1.5   3.0

g_num
1 5  2 5  3 5  4 5

nr,(k,nf(:,k),i=1,nr)
4
1 0 0 0  2 0 0 0  3 0 0 0  4 0 0 0

loaded_nodes,(k,loads(nf(:,k))
1
5  20.0  -20.0  30.0

fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
1
5  2  -0.0005
```

Figure 4.9 (*Continued from page 120*)

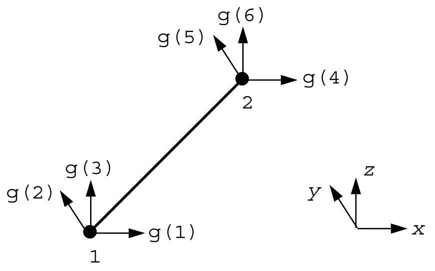


Figure 4.10 Node and freedom numbering for 3D rod elements

```
There are    3 equations and the skyline storage is    6

 Node   Displacement(s)
   1  0.0000E+00  0.0000E+00  0.0000E+00
   2  0.0000E+00  0.0000E+00  0.0000E+00
   3  0.0000E+00  0.0000E+00  0.0000E+00
   4  0.0000E+00  0.0000E+00  0.0000E+00
   5  0.2569E-03 -0.5000E-03  0.7614E-04

Element Actions
   1  0.1298E+00  0.9735E-01  0.1947E+00 -0.1298E+00 -0.9735E-01 -0.1947E+00
      Axial force = -0.2534E+00
   2 -0.1082E+02  0.2163E+02 -0.4327E+02  0.1082E+02 -0.2163E+02  0.4327E+02
      Axial force =  0.4957E+02
   3  0.1789E+01  0.5963E+00 -0.3578E+01 -0.1789E+01 -0.5963E+00  0.3578E+01
      Axial force =  0.4045E+01
   4 -0.1110E+02  0.2775E+01  0.1665E+02  0.1110E+02 -0.2775E+01 -0.1665E+02
      Axial force = -0.2021E+02
```

Figure 4.11 Results from second Program 4.2 example

the data provides a load *and* a fixed displacement at the same freedom, the value of the load is immaterial, and the displacement always takes precedence.

Program 4.3 Analysis of elastic beams using 2-node beam elements (elastic foundation optional).

```
PROGRAM p43
!-------------------------------------------------------------------------
! Program 4.3 Analysis of elastic beams using 2-node beam elements
!             (elastic foundation optional).
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndof=4,nels,neq,nod=2,      &
   nodof=2,nn,nprops,np_types,nr
 REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),kdiag(:),nf(:,:),no(:),      &
   node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::action(:),eld(:),ell(:),km(:,:),kv(:),loads(:),   &
   mm(:,:),prop(:,:),value(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,nprops,np_types; nn=nels+1
 ALLOCATE(g(ndof),num(nod),nf(nodof,nn),etype(nels),ell(nels),eld(ndof),  &
   km(ndof,ndof),mm(ndof,ndof),action(ndof),g_g(ndof,nels),              &
   prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----------------------loop the elements to find global array sizes------
 elements_1: DO iel=1,nels
   num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------global stiffness matrix assembly------------------
 kv=zero
 elements_2: DO iel=1, nels
   CALL beam_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
   mm=zero; IF(nprops>1)CALL beam_mm(mm,prop(2,etype(iel)),ell(iel))
   CALL fsparv(kv,km+mm,g,kdiag)
 END DO elements_2
!-----------------------read loads and/or displacements-------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),no(fixed_freedoms),                      &
     sense(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
!-----------------------equation solution --------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 WRITE(11,'(/A)')"  Node Translation Rotation"
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(nf(:,k)); END DO
!-----------------------retrieve element end actions----------------------
 WRITE(11,'(/A)')" Element Force       Moment      Force       Moment"
 elements_3: DO iel=1,nels
   CALL beam_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
   mm=zero; IF(nprops>1)CALL beam_mm(mm,prop(2,etype(iel)),ell(iel))
   eld=loads(g); action=MATMUL(km+mm,eld)
   WRITE(11,'(I5,4E12.4)')iel,action
 END DO elements_3
STOP
END PROGRAM p43
```

New dynamic real arrays:

`mm`	element "mass" matrix

This program has much in common with Program 4.1 for rod elements. A line of beam elements of different stiffnesses and lengths, optionally resting on an elastic foundation, can be analysed for any combination of transverse and/or moment loading.

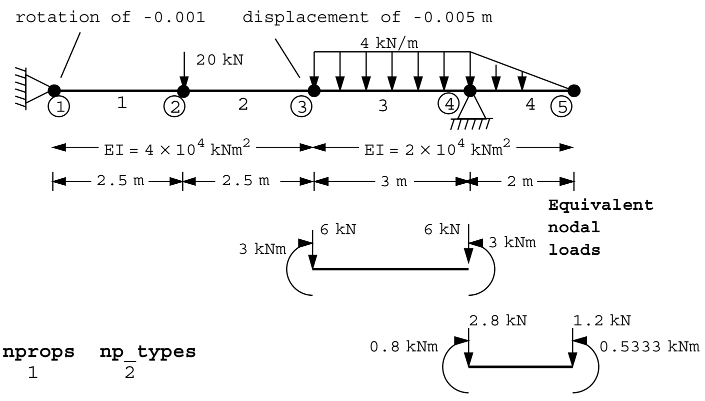
The inclusion of an elastic foundation is signalled by reading `nprops` as 2, indicating that two properties, namely the beam flexural stiffness *EI* and the foundation stiffness *k*, must be read into the properties matrix `prop`. If `nprops` is read as 1 however, only one property, the flexural stiffness *EI* is required as data and the analysis is of a simple beam.

The beam element stiffness matrix is provided by subroutine `beam_km`, and the foundation stiffness matrix, closely related to the element "mass" matrix (see Section 2.4.2), by subroutine `beam_mm`. The real array `mm` is included in the program to hold the foundation stiffness matrix. As in Program 4.1, the length of each element is read into the real vector `ell`.

The first example problem shown in Figure 4.12, represents a non-uniform beam subjected to a combination of nodal loads and fixed displacements. Node 1 is to be rotated clockwise by 0.001 and node 3 is to be translated vertically downwards by 0.005. In addition, a vertical force of 20 acts at node 2, a uniformly distributed load of 4/unit length acts between nodes 3 and 4, and a linearly decreasing load of 4/unit length to zero acts between nodes 4 and 5.

At each node, 2 degrees of freedom are possible, a vertical translation and a rotation, in that order. The global node numbering reads from left to right, and at the element level, node one is always to the left and node two to the right as shown in Figure 4.13. Each element has 4 degrees of freedom taken in the order w_1, θ_1, w_2, and θ_2. The nodal freedom numbering associated with each element, accounting for any restraints, is as usual contained in the "steering" vector `g`. Thus, with reference to Figures 4.12 and 4.13, the steering vector for element 1 would be $[0\ 1\ 2\ 3]^T$ and for element 2, $[2\ 3\ 4\ 5]^T$, and so on. It should be noted that in the "penalty" or "stiff spring" technique, the freedoms to be fixed are assembled into the global stiffness matrix in the usual way prior to augmenting the appropriate diagonal and force terms (see Section 3.6).

It should also be pointed out that when freedoms are fixed to zero, as is common at the boundaries of solid or structural analyses, the user has the choice of fixing them through boundary condition data, using `nr`, in which case the equations are never assembled, or through the "stiff spring" (or "penalty") technique. The data in Figure 4.12 uses the former strategy, however Figure 4.14 shows an alternative data set that achieves essentially the



```
nels  nprops  np_types
4       1       2

props(ei)
4.0e4  2.0e4

etype
1  1  2  2

ell
2.5  2.5  3.0  2.0

nr,(k,nf(:,k),i=1,nr)
2
1 0 1  4 0 1

loaded_nodes,(k,loads(nf(:,k))
4
2 -20.0   0.0      3  -6.0  -3.0
4  -8.8   2.2      5  -1.2   0.5333

fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
2
1 2 -0.001  3 1 -0.005
```

Figure 4.12 Mesh and data for first Program 4.3 example

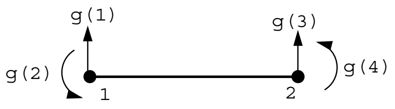


Figure 4.13 Node and freedom numbering for beam elements

same results using “stiff springs”. An advantage of the “stiff spring” approach for fixing all boundary conditions, is that all nodes retain their full complement of freedoms, leading to a simpler freedom numbering system. Disadvantages are that there are greater memory requirements with more equations needing to be solved and the chances of numerical difficulties are higher.

Nodal loading in the context of beam analysis can take the form of either point loads or moments. In the analysis, loading can only be applied at the nodes, so if forces or moments are required between nodes a set of equivalent nodal loads must be derived for application to the mesh. These equivalent nodal loads can be found by computing the shear forces and moments that would have been generated at each node if the element had been fully

```
nels  nprops  np_types
4       1       2

props(ei)
4.0e4  2.0e4

etype
1  1  2  2

ell
2.5  2.5  3.0  2.0

nr
0

loaded_nodes,(k,loads(nf(:,k))
4
2 -20.0   0.0      3  -6.0  -3.0
4  -8.8   2.2      5  -1.2   0.5333

fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
4
1 1  0.0    1 2 -0.001
3 1 -0.005  4 1  0.0
```

Figure 4.14 Alternative data for first Program 4.3 example without using ‘nr’ data

```
There are    8 equations and the skyline storage is   21

Node Translation Rotation
  1  0.0000E+00 -0.1000E-02
  2 -0.3579E-02 -0.1301E-02
  3 -0.5000E-02  0.2051E-03
  4  0.0000E+00  0.2410E-02
  5  0.4713E-02  0.2343E-02

Element Force       Moment       Force       Moment
  1  0.2157E+02  0.3178E+02 -0.2157E+02  0.2214E+02
  2  0.1569E+01 -0.2214E+02 -0.1569E+01  0.2606E+02
  3 -0.9577E+01 -0.2906E+02  0.9577E+01  0.3333E+00
  4  0.1200E+01  0.1867E+01 -0.1200E+01  0.5333E+00
```

Figure 4.15 Results from first Program 4.3 example

“encastré” at both ends (the fixed end moments and shear forces). The signs of these fixed end values are then reversed and applied to the nodes of the element in the actual problem (see e.g. Przemieniecki, 1968). If loads or moments are required at the nodes themselves, they are applied directly to the node without any further manipulation. If point loads are to be applied to a beam it is recommended that a node be placed at that location to avoid the need for unnecessary “fixed end” calculations.

In the example of Figure 4.12, elements 3 and 4 support distributed loads and the appropriate equivalent nodal loads to be applied are shown beneath their respective elements.

When the computed results shown in Figure 4.15 are examined, it will be seen that the fixed freedoms have the expected values. This is confirmed by the rotation at node 1 of −0.001 (clockwise) and the translation at node 3 of −0.005. Of the remaining displacements it is seen that the rotation at node 3, for example, equals 0.000205 (anti-clockwise).

In order to compute the actual moments and shear forces in the beam, the equivalent nodal loads must be subtracted from the corresponding “action” vector printed for each element. This is of course only necessary for those elements that involved loading between the nodes. For example, the moments at the nodes in this example for each of the elements are given as follows:

Element 1

$$M_1 = 31.78$$
$$M_2 = 22.14$$

Element 2

$$M_1 = -22.14$$
$$M_2 = 26.06$$

Element 3

$$M_1 = -29.06 + 3.00 = -26.06$$
$$M_2 = \ \ 0.33 - 3.00 = \ \ -2.67$$

Element 4

$$M_1 = 1.87 + 0.80 = 2.67$$
$$M_2 = 0.53 - 0.53 = 0.00$$

Internal equilibrium is maintained, for example M_2 of element 1 is equal and opposite to M_1 of element 2, and so on. A bending moment diagram for the beam based on these values is given in Figure 4.16.

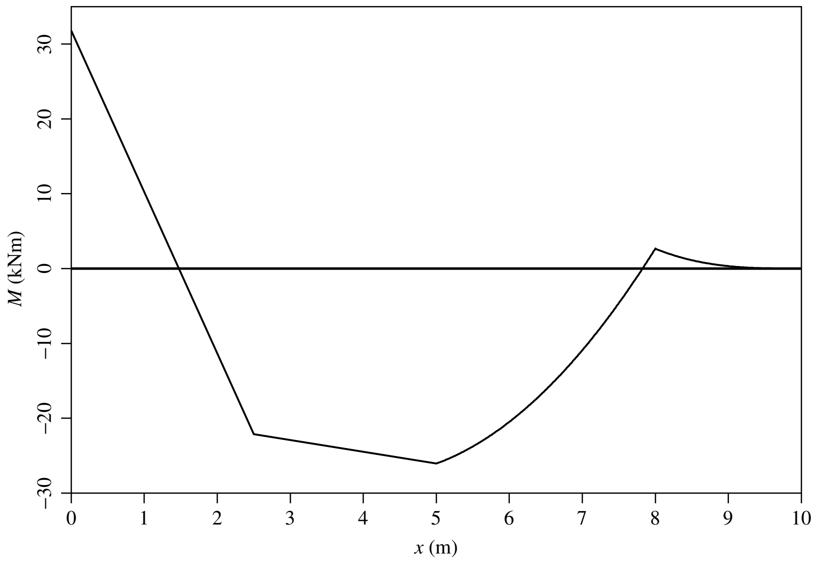
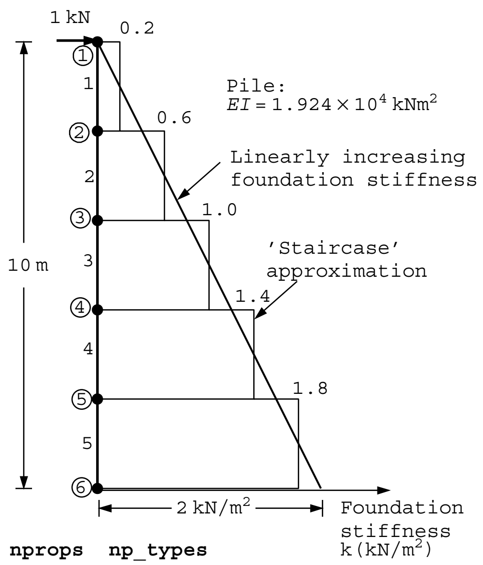


Figure 4.16 Bending moment diagram from first Program 4.3 example

A second example of the use of Program 4.3 is illustrated in Figure 4.17 and represents a laterally loaded pile which is to be modelled as a "beam on an elastic foundation", with


1 kN
0.2
0.6
1.0
1.4
1.8
Pile:
$EI = 1.924 \times 10^4$ kNm^2
Linearly increasing foundation stiffness
'Staircase' approximation
10 m
①
②
③
④
⑤
⑥
1
2
3
4
5
2 kN/m^2
Foundation stiffness k(kN/m^2)

```
nels   nprops   np_types
5        2        5
props(ei,k)
1.924e4  0.2   1.924e4  0.6
1.924e4  1.0   1.924e4  1.4
1.924e4  1.8
etype
1  2  3  4  5
ell
2.0  2.0  2.0  2.0  2.0
nr
0
loaded_nodes,(k,loads(nf(:,k))
1
1  1.0  0.0
fixed_freedoms
0
```

Figure 4.17 Mesh and data for second Program 4.3 example

a linearly increasing soil or foundation stiffness. The pile has a constant flexural stiffness of $EI = 1.924 \times 10^4$ kNm^2, and the foundation stiffness increases from zero at the ground surface to 2 kN/m^2 at a depth of 10 m. The pile is modelled by 5 beam elements, and the soil stiffness is approximated by a step function based on the stiffness at the mid-point of each element. The data file provides `nprops=2` to signify the presence of an "elastic foundation", and `np_types=5` since each element is supported by soil with a different stiffness value. The composite beam/foundation global stiffness matrix in this case involves the assembly of the sum of the element stiffness and "mass" matrices, `km` and `mm` respectively.

The remainder of the program follows a familiar course. Forces and/or fixed displacements are read, and, following equation solution, the global nodal displacements and rotations are obtained. In the post-processing phase, the element nodal displacements `eld` are retrieved, as are the element stiffness and "mass" matrices. The product of `eld` and the

```
There are   12 equations and the skyline storage is   38

 Node Translation Rotation
   1  0.8559E+00 -0.1148E+00
   2  0.6263E+00 -0.1147E+00
   3  0.3971E+00 -0.1145E+00
   4  0.1683E+00 -0.1143E+00
   5 -0.6008E-01 -0.1141E+00
   6 -0.2883E+00 -0.1141E+00

Element Force       Moment       Force       Moment
   1  0.1000E+01  0.3638E-11 -0.7036E+00  0.1688E+01
   2  0.7036E+00 -0.1688E+01 -0.8960E-01  0.2436E+01
   3  0.8960E-01 -0.2436E+01  0.4757E+00  0.1973E+01
   4 -0.4757E+00 -0.1973E+01  0.6271E+00  0.7640E+00
   5 -0.6271E+00 -0.7640E+00 -0.9095E-12 -0.9095E-12
```

Figure 4.18 Results from second Program 4.3 example

element composite stiffness km+mm gives the element "actions" which holds the element shear forces and moments.

The pile is supported by the foundation only, so no additional boundary conditions are needed (nr=0). In this example, a unit horizontal load has been applied to the top of the pile.

The computed results in Figure 4.18 show the horizontal translation and rotation at each of the nodes. It is seen that the horizontal translation of node 1 is given as 0.856 which is in reasonable agreement with the analytical solution to this problem from Hetenyi (1946).

Program 4.4 Analysis of elastic rigid-jointed frames using 2-node beam/rod elements in two or three dimensions.

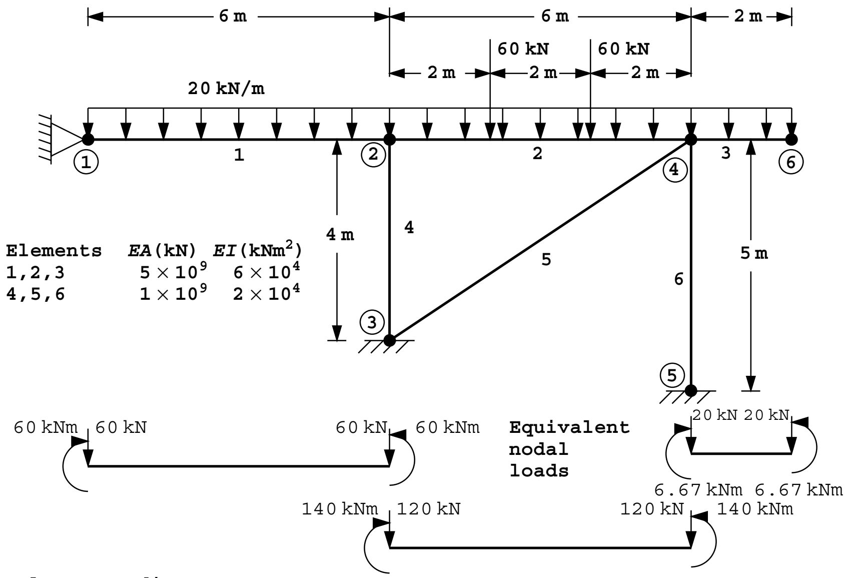
```
PROGRAM p44
!-------------------------------------------------------------------------
! Program 4.4 Analysis of elastic rigid-jointed frames using 2-node
!             beam/rod elements in 2- or 3-dimensions.
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim,ndof,nels,neq,nod=2,   &
   nodof,nn,nprops,np_types,nr
 REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::action(:),coord(:,:),eld(:),gamma(:),g_coord(:,:),&
   km(:,:),kv(:),loads(:),prop(:,:),value(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,nn,ndim,nprops,np_types
 IF(ndim==2)nodof=3; IF(ndim==3)nodof=6; ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),km(ndof,ndof),coord(nod,ndim),g_coord(ndim,nn),    &
   eld(ndof),action(ndof),g_num(nod,nels),num(nod),g(ndof),gamma(nels),   &
   g_g(ndof,nels),prop(nprops,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype;
 IF(ndim==3)READ(10,*)gamma
 READ(10,*)g_coord; READ(10,*)g_num
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----------------------loop the elements to find global array sizes------
```

```
 elements_1: DO iel=1,nels
   num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------global stiffness matrix assembly------------------
 kv=zero
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!-----------------------read loads and/or displacements-------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),no(fixed_freedoms),                      &
     sense(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
!-----------------------equation solution --------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 WRITE(11,'(/A)')  "  Node   Displacements and Rotation(s)"
 DO k=1,nn; WRITE(11,'(I5,6E12.4)')k,loads(nf(:,k)); END DO
!-----------------------retrieve element end actions----------------------
 WRITE(11,'(/A)')" Element Actions"
 elements_3: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
   eld=loads(g); action=MATMUL(km,eld)
   IF(ndim<3)THEN; WRITE(11,'(I5,6E12.4)')iel,action; ELSE
     WRITE(11,'(I5,6E12.4)')iel,   action(1: 6)
     WRITE(11,'(A,6E12.4)')"     ",action(7:12)
   END IF
 END DO elements_3
STOP
END PROGRAM p44
```

New dynamic real arrays:

`gamma`	rotation of element about local axis

The first three programs in this chapter were concerned only with 1D elements which could sustain either axial loads (rod elements) or transverse loading and moments (beam elements). It is much more common to encounter structures made up of members arbitrarily inclined and attached to one another. Loading of such structures results in displacements due to both axial and bending effects, although the former is often ignored in many approximate methods. The beam–rod element stiffness matrix used by this program is formed by superposing the beam and rod stiffness matrices described in earlier programs in this chapter. The program described in this section can analyse two- or three-dimensional framed structures, with the element stiffness matrix formed by the library subroutine `rigid_jointed`.



```
nels   nn   ndim   nprops   np_types
6      6    2      2        2

prop(ea,ei)
5.0e9  6.0e4
1.0e9  2.0e4

etype
1  1  1  2  2  2

g_coord
 0.0  0.0    6.0  0.0    6.0 -4.0
12.0  0.0   12.0 -5.0   14.0  0.0

g_num
1 2  2 4  4 6  3 2  3 4  5 4

nr,(k,nf(:,k),i=1,nr)
3
1 0 0 1  3 0 0 0  5 0 0 0

loaded_nodes,(k,loads(nf(:,k))
4
1  0.0  -60.0  -60.0     2  0.0 -180.0  -80.0
4  0.0 -140.0  133.33    6  0.0  -20.0    6.67

fixed_freedoms
0
```

Figure 4.19 Mesh and data for first Program 4.4 example

The first example analysed by Program 4.4 is shown in Figure 4.19 and is a 2D rigid-jointed frame subjected to distributed loads and point loads. In 2D, the elements have 6 degrees of freedom as shown in Figure 4.20. At each node there are two translational freedoms in x- and y- and a rotation (in that order). The nodal freedom numbering associated with each element, accounting for any restraints, is as usual contained in the "steering" vector g. Thus, with reference to Figures 4.19 and 4.20, the steering vector for element 1 would be $[0\ 0\ 1\ 2\ 3\ 4]^T$ and for element 2, $[2\ 3\ 4\ 5\ 6\ 7]^T$, and so on. The data organisation

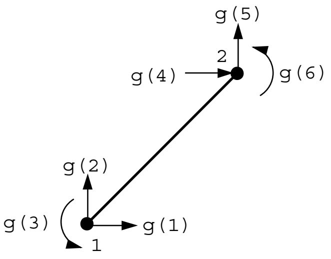


Figure 4.20 Node and freedom numbering for 2D beam–rod elements

```
 There are   10 equations and the skyline storage is   40

  Node   Displacements and Rotation(s)
     1  0.0000E+00  0.0000E+00 -0.1025E-02
     2  0.3645E-07 -0.8319E-06 -0.9497E-03
     3  0.0000E+00  0.0000E+00  0.0000E+00
     4  0.6435E-07 -0.6283E-06  0.1774E-02
     5  0.0000E+00  0.0000E+00  0.0000E+00
     6  0.6435E-07  0.2880E-02  0.1329E-02

 Element Actions
     1 -0.3038E+02 -0.1975E+02 -0.6000E+02  0.3038E+02  0.1975E+02 -0.5849E+02
     2 -0.2325E+02  0.8238E+01 -0.2519E+01  0.2325E+02 -0.8238E+01  0.5195E+02
     3  0.0000E+00  0.2000E+02  0.3333E+02  0.0000E+00 -0.2000E+02  0.6670E+01
     4  0.7123E+01  0.2080E+03 -0.9497E+01 -0.7123E+01 -0.2080E+03 -0.1899E+02
     5  0.3177E+02  0.2610E+02  0.9839E+01 -0.3177E+02 -0.2610E+02  0.1968E+02
     6 -0.8513E+01  0.1257E+03  0.1419E+02  0.8513E+01 -0.1257E+03  0.2838E+02
```

Figure 4.21 Results from first Program 4.4 example

is similar to the pin-jointed frame analysis described in Program 4.2. The first line of data provides the number of elements (`nels`), the number of nodes (`nn`), the dimensionality (`ndim`), the number of properties `nprops` and the number of property types `np_types`.

In this program, the number of material properties required depends on the dimensionality, so the data now includes input to the integer `nprops` which indicates the number of material properties required for each property type. In a 2D frame problem, there are two material properties required (`EA` and `EI`), so `nprops = 2`. The material property values for each type are then read into the two-dimensional array `prop`.

The material property data is followed by the `etype` vector (if needed), the global nodal coordinates (`g_coord`) and the element node numbering (`g_num`). The loading on the nodes is calculated using the equivalent nodal loads approach described previously for the first example with Program 4.3, and these values are shown for each individual element in Figure 4.19. There are no `fixed_freedoms` in this example. The results shown in Figure 4.21 indicate that the rotation at node 1 for example is −0.001025 (clockwise). The `action` vectors for elements 4, 5, and 6 are correct as printed, however for elements 1,2, and 3 the equivalent nodal loads must be subtracted, for example

Element 1

$$\begin{aligned} F_{x1} &= -30.38 + 0.00 = -30.38 \\ F_{y1} &= -19.75 + 60.00 = 40.25 \end{aligned}$$

$$\begin{aligned}
M_1 &= -60.00 + 60.00 = 0.00\\
F_{x1} &= 30.38 + 0.00 = 30.38\\
F_{y1} &= 19.75 + 60.00 = 79.75\\
M_2 &= -58.49 - 60.00 = -118.49
\end{aligned}$$

Element 2

$$\begin{aligned}
F_{x1} &= -23.25 + 0.00 = -23.25\\
F_{y1} &= 8.24 + 120.00 = 128.24\\
M_1 &= -2.52 + 140.00 = 137.48\\
F_{x1} &= 23.25 + 0.00 = 23.25\\
F_{y1} &= -8.24 + 120.00 = 111.76\\
M_2 &= 51.95 - 140.00 = -88.05
\end{aligned}$$

Element 3

$$\begin{aligned}
F_{x1} &= 0.00 + 0.00 = 0.00\\
F_{y1} &= 20.00 + 20.00 = 40.00\\
M_1 &= 33.33 + 6.67 = 40.00\\
F_{x1} &= 0.00 + 0.00 = 0.00\\
F_{y1} &= -20.00 + 20.00 = 0.00\\
M_2 &= 6.67 - 6.67 = 0.00
\end{aligned}$$

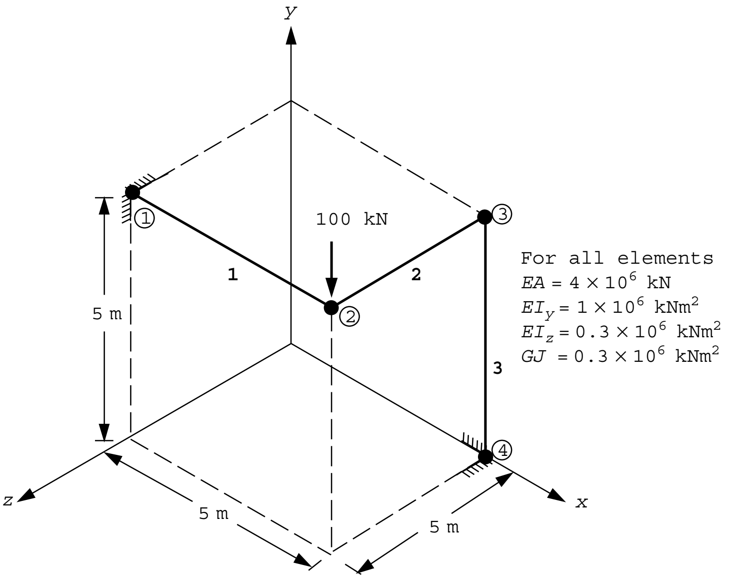
Moment equilibrium is established by adding the moments from the appropriate end of all elements coming into the joint.

The second example to be analysed by Program 4.4 is shown in Figure 4.22 and represents a 3D rigid-jointed frame subjected to a vertical point load of -100.0. In 3D, the elements have 12 degrees of freedom as shown in Figure 4.23. At each node there are three translational freedoms in x-, y-, and z-, and three rotations about each of the global axes. The extension to 3D is conceptually simple, but considerably more care is required in the preparation of data and attention to sign conventions. The data organisation is virtually the same as in the previous example, except `ndim` is set to 3 in the data.

In addition to the axial stiffness (EA), 3D involves the flexural stiffness about the element's local y' and z' axes (EI_y and EI_z respectively) and a torsional stiffness (GJ), thus `nprops` is 4. The local coordinate x' defines the long axis of the element. The relationship between the global axes (x, y, z) and local axes (x', y', z') must be considered for 3D space frames because in addition to the six coordinates that define the position of each node of the element in space, a seventh rotational "coordinate" γ must be read in as data. The additional real vector `gamma` is provided to hold this information (in degrees) for each element.

For the purposes of data preparation, a "vertical" element is defined as one which lies parallel to the global y axis. For non-vertical elements the angle γ is defined as the rotation of the element about its local x' axis as shown in Figure 4.24. For "vertical" elements however, γ is defined as the angle between the global z axis and the local z' axis, measured towards the global x axis as shown in Figure 4.25. For "vertical" elements it is essential that the local x' axis points in the same direction as the global y axis.

Returning to the example problem, it is necessary to establish the local coordinate system for each element as shown in Figure 4.26. As indicated in Figure 4.23, the positive local x' direction is defined by moving from node 1 to node 2, so this is also the order in which the element nodal numbering must be given in the data.


y
100 kN
1
2
3
5 m
5 m
5 m
z
x
For all elements
$EA = 4 \times 10^6$ kN
$EI_y = 1 \times 10^6$ kNm^2
$EI_z = 0.3 \times 10^6$ kNm^2
$GJ = 0.3 \times 10^6$ kNm^2

```
nels   nn   ndim   nprops   np_types
3      4    3       4         1

prop(ea,eiy,eiz,gj)
4.0e6  1.0e6  0.3e6  0.3e6

etype(not needed)

gamma
0.0   0.0  90.0

g_coord
0.0  5.0  5.0    5.0  5.0  5.0
5.0  5.0  0.0    5.0  0.0  0.0

g_num
1 2  3 2  4 3

nr,(k,nf(:,k),i=1,nr)
2
1 0 0 0 0 0 0   4 0 0 0 0 0 0

loaded_nodes,(k,loads(nf(:,k))
1
2 0.0 -100.0  0.0  0.0  0.0  0.0

fixed_freedoms
0
```

Figure 4.22 Mesh and data for second Program 4.4 example

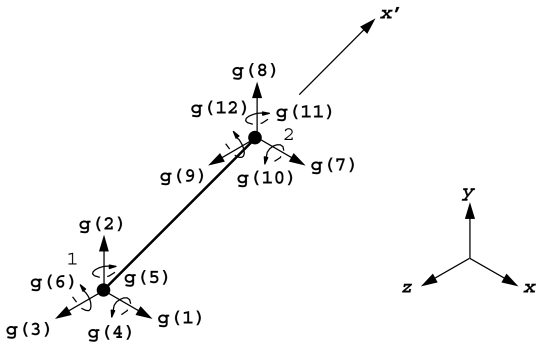


Figure 4.23 Node and freedom numbering for 3D beam–rod elements

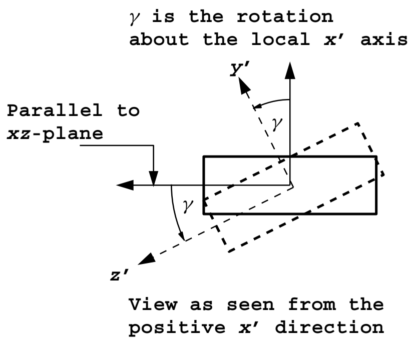

γ is the rotation about the local x' axis
y'
γ
Parallel to xz-plane
γ
z'
View as seen from the positive x' direction

Figure 4.24 Transformation angle for "non-vertical" elements

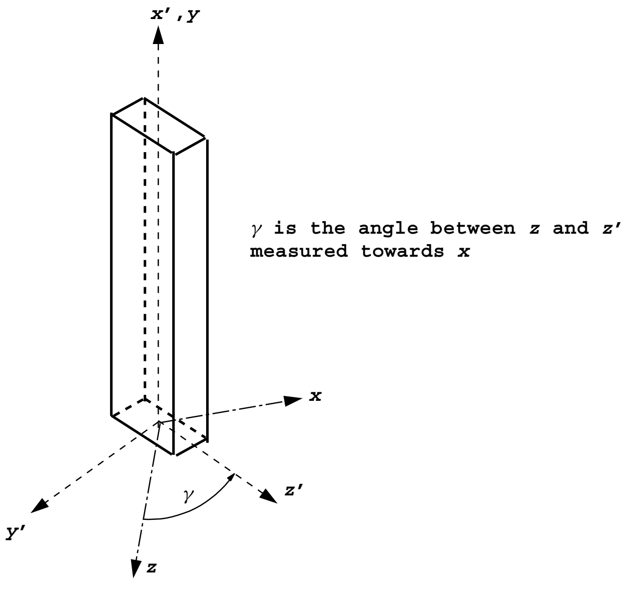

x',y
γ is the angle between z and z' measured towards x
x
z'
γ
y'
z

Figure 4.25 Transformation angle for "vertical" elements

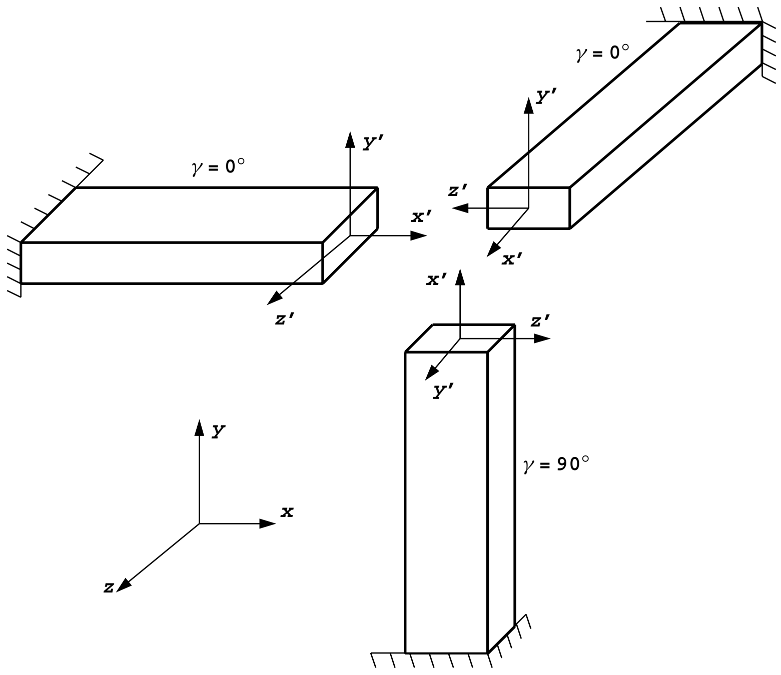


Figure 4.26 Element local coordinate systems

```
There are   12 equations and the skyline storage is   78

 Node   Displacements and Rotation(s)
   1  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
   2 -0.3039E-05 -0.5997E-02  0.8769E-03  0.1129E-02 -0.2360E-03 -0.1514E-02
   3  0.9571E-03 -0.4536E-04  0.9113E-03  0.7470E-03 -0.1582E-03 -0.3727E-03
   4  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00

Element Actions
   1  0.2431E+01  0.6371E+02 -0.2754E+02 -0.6777E+02  0.1160E+03  0.2501E+03
     -0.2431E+01 -0.6371E+02  0.2754E+02  0.6777E+02  0.2165E+02  0.6846E+02
   2 -0.2431E+01  0.3629E+02  0.2754E+02 -0.1137E+03  0.9490E+01  0.6846E+02
      0.2431E+01 -0.3629E+02 -0.2754E+02 -0.6777E+02 -0.2165E+02 -0.6846E+02
   3 -0.2431E+01  0.3629E+02  0.2754E+02  0.2403E+02  0.9490E+01  0.8062E+02
      0.2431E+01 -0.3629E+02 -0.2754E+02  0.1137E+03 -0.9490E+01 -0.6846E+02
```

Figure 4.27 Results from second Program 4.4 example

In the example of Figure 4.22, elements 1 and 2 both have their local z' axes parallel to the global xz plane, thus there has been no rotation of these non-vertical elements and γ is set to zero. For vertical element 3 however, γ is set to 90°, which is the angle between the global z and local z' axes measured towards the global x axis.

The results of the analysis given in Figure 4.27 indicate that the vertical deflection under the load is -0.005997. As a check on equilibrium under the applied loading of -100.0, the F_{y1} component of the `action` vector at the built-in end (node 1) of each of elements 1 and 3 is 63.71 and 39.29 respectively.

Program 4.5 Analysis of elastic–plastic beams or rigid-jointed frames using 2-node beam or beam/rod elements in one, two or three dimensions.

```
PROGRAM p45
!-------------------------------------------------------------------------
! Program 4.5 Analysis of elasto-plastic beams or rigid-jointed frames
!             using 2-node beam or beam/rod elements in 1-, 2- or
!             3-dimensions
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,incs,iters,iy,k,limit,loaded_nodes,ndim,ndof,nels,neq,    &
   nod=2,nodof,nn,nprops,np_types,nr
 REAL(iwp)::tol,total_load,zero=0.0_iwp; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
  node(:),num(:)
 REAL(iwp),ALLOCATABLE::action(:),bdylds(:),coord(:,:),dload(:),eld(:),   &
   eldtot(:),gamma(:),g_coord(:,:),holdr(:,:),km(:,:),kv(:),loads(:),     &
   oldis(:),prop(:,:),react(:),val(:,:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,nn,ndim,nprops,np_types
 IF(ndim==1)nodof=2; IF(ndim==2)nodof=3; IF(ndim==3)nodof=6; ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),km(ndof,ndof),coord(nod,ndim),g_coord(ndim,nn),    &
   eld(ndof),action(ndof),g_num(nod,nels),num(nod),g(ndof),gamma(nels),   &
   g_g(ndof,nels),holdr(ndof,nels),react(ndof),prop(nprops,np_types),     &
   etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype;
 IF(ndim==3)READ(10,*)gamma
 READ(10,*)g_coord; READ(10,*)g_num
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),eldtot(0:neq),bdylds(0:neq),oldis(0:neq))
!-----------------------loop the elements to find global array sizes------
 kdiag=0
 elements_1: DO iel=1,nels
   num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------global stiffness matrix assembly------------------
 holdr=zero; kv=zero
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
 READ(10,*)loaded_nodes
 ALLOCATE(node(loaded_nodes),val(loaded_nodes,nodof))
 READ(10,*)(node(i),val(i,:),i=1,loaded_nodes); READ(10,*)limit,tol,incs
 ALLOCATE(dload(incs)); READ(10,*)dload
!-----------------------equation factorisation----------------------------
 CALL sparin(kv,kdiag); total_load=zero
!-----------------------load increment loop-------------------------------
 load_incs: DO iy=1,incs
   total_load=total_load+dload(iy); WRITE(*,'(/,A,I5)')" load step",iy
   WRITE(11,'(/A,i3,A,E12.4)')" Load step",iy,"    Load factor ",total_load
   oldis=zero; iters=0
   its: DO
     iters=iters+1; WRITE(*,*)"iteration no",iters; loads=zero
     DO i=1,loaded_nodes; loads(nf(:,node(i)))=dload(iy)*val(i,:); END DO
     loads=loads+bdylds; bdylds=zero
!-----------------------forward/back-substitution and check convergence---
     CALL spabac(kv,loads,kdiag); loads(0)=zero
     CALL checon(loads,oldis,tol,converged)
!-----------------------inspect moments in all elements-------------------
     elements_3: DO iel=1,nels
       num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
       CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
       eld=loads(g); action=MATMUL(km,eld); react=zero
!------------------if plastic moments exceeded generate correction vector-
       IF(limit/=1)THEN
         CALL hinge(coord,holdr,action,react,prop,iel,etype,gamma)
         bdylds(g)=bdylds(g)-react; bdylds(0)=zero
       END IF
!-----------------------at convergence update element reactions-----------
       IF(iters==limit.OR.converged)                                      &
         holdr(:,iel)=holdr(:,iel)+react(:)+action(:)
     END DO elements_3
     IF(iters==limit.OR.converged)EXIT its
   END DO its
   eldtot=loads+eldtot;
   WRITE(11,'(A)')  "  Node   Displacement(s) and Rotation(s)"
   DO i=1,loaded_nodes
     WRITE(11,'(I5,6E12.4)')node(i),eldtot(nf(:,node(i)))
   END DO
   WRITE(11,'(A,I5,A)')" Converged in",iters," iterations"
   IF(iters==limit.AND.limit/=1)EXIT load_incs
 END DO load_incs
STOP
END PROGRAM p45
```

New scalar integers:

`incs`	number of load increments
`iters`	counts plastic iterations
`iy`	counts load increments
`limit`	plastic iteration ceiling

New scalar reals:

`tol`	plastic convergence tolerance
`total_load`	running total of `dload` (= λ)

Scalar logical:

`converged`	set to `.TRUE.` if algorithm has converged

New dynamic real arrays:

`bdylds`	internal correction “forces” to redistribute moments
`dload`	load increment values
`eldtot`	keeps a running total of nodal displacements

`holdr`	holds element "actions" at convergence
`oldis`	nodal displacements from previous iteration
`react`	element self-equilibrating "correction" vector
`val`	nodal load weightings

This program is an extension of the preceding Program 4.4 in which a limit is placed on the maximum moment that any member can sustain. As loads on the structure are increased, plastic hinges form progressively and "collapse" occurs when a mechanism develops. This program is currently limited to load control analysis.

This is the first example in the book of a non-linear analysis in which the moment-curvature behaviour of the members is assumed to be elastic-perfectly plastic as shown in Figure 4.28. To deal with this non-linearity, an iterative approach is used to find the nodal displacements and element "actions" under a given set of applied loads. Moments in excess of their plastic limits are re-distributed to other joints which still have reserves of moment carrying capacity. Convergence of the iterative process is said to have occurred when, within certain tolerances, moments at the element nodes nowhere exceed their limiting plastic values, and the internal "actions" are in equilibrium with the applied external loads.

The conventional approach for tackling this type of problem is progressively to modify the global stiffness matrix as joints reach the plastic limit. The modification is necessary because a plastic joint is replaced by a pin joint with the appropriate plastic moment applied. In the method shown in the structure chart in Figure 4.29, the global stiffness

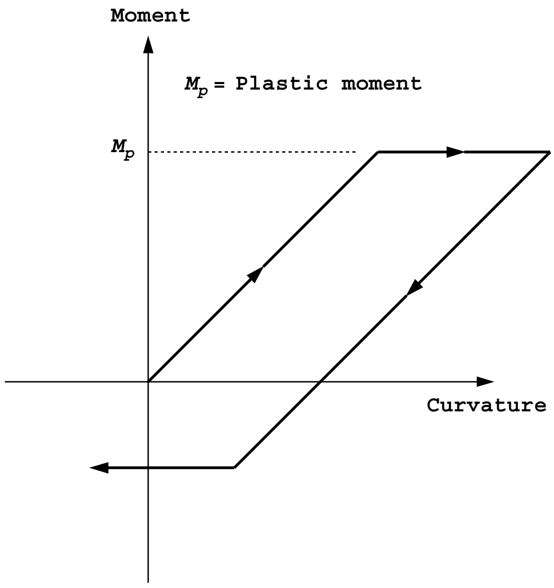
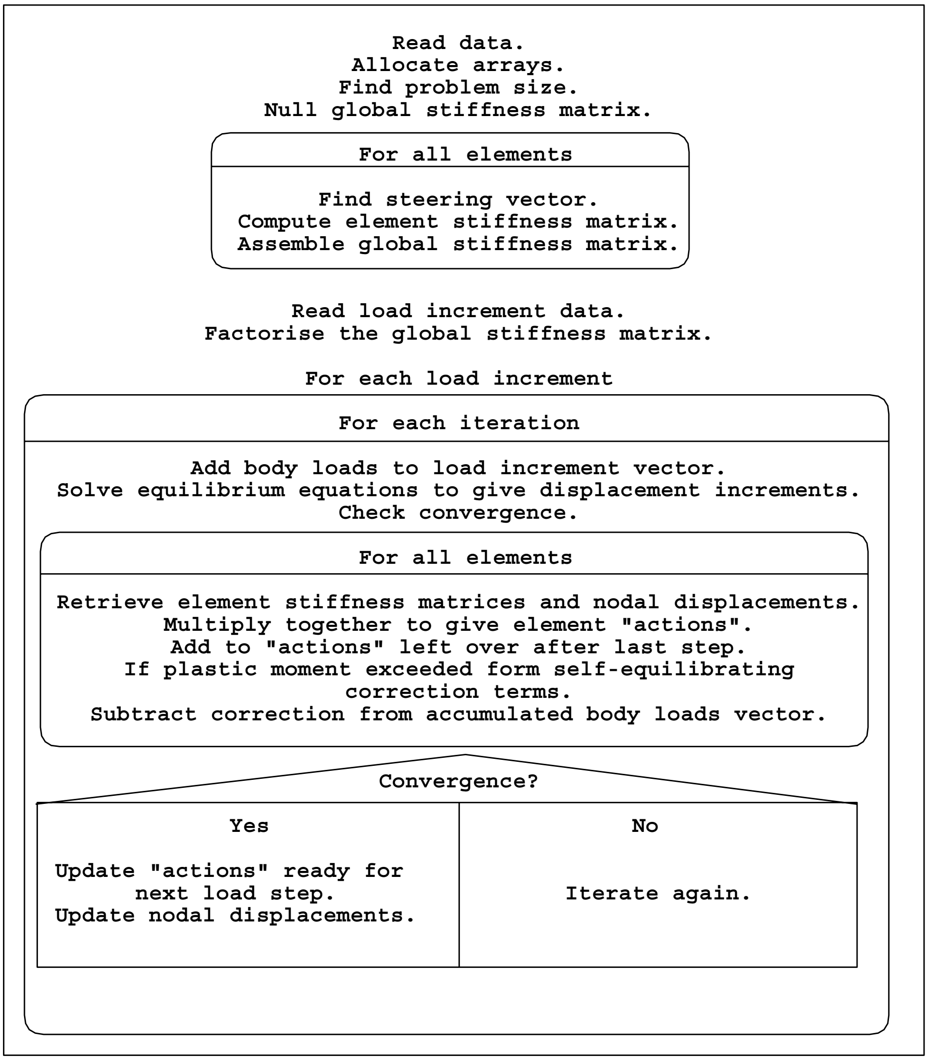


Figure 4.28 Elastic-perfectly plastic moment-curvature relationship



```
Read data.
Allocate arrays.
Find problem size.
Null global stiffness matrix.
For all elements
    Find steering vector.
    Compute element stiffness matrix.
    Assemble global stiffness matrix.
Read load increment data.
Factorise the global stiffness matrix.
For each load increment
    For each iteration
        Add body loads to load increment vector.
        Solve equilibrium equations to give displacement increments.
        Check convergence.
        For all elements
            Retrieve element stiffness matrices and nodal displacements.
            Multiply together to give element "actions".
            Add to "actions" left over after last step.
            If plastic moment exceeded form self-equilibrating
                correction terms.
            Subtract correction from accumulated body loads vector.
        Convergence?
            Yes
                Update "actions" ready for
                    next load step.
                Update nodal displacements.
            No
                Iterate again.
```

Figure 4.29 Structure chart for Program 4.5

matrix is formed once only, with the non-linearity introduced by iteratively modifying the applied forces on the structure until convergence is achieved. For greater detail of this particular algorithm the reader is referred to Griffiths (1988). Similar procedures are utilised in Chapter 6 in relation to elastic–plastic solids.

When a problem involves a constant left hand side matrix and multiple right hand side vectors, the benefits of splitting the solution of the equilibrium equations into two stages, namely factorisation performed once (using subroutine `sparin`), and a forward and back substitution performed at each iteration for each new right-hand side (using subroutine spabac) become clear.

Material properties in Program 4.5 must now include both elastic properties and plastic moment values for all members. For 1D beams or 2D frames, only one plastic moment (M_p) is read, thus nprops is 2 in 1D and 3 in 2D. For 3D space frames, however, three plastic moments, (M_{py}, M_{pz}, and M_{px} in that order) are read, thus nprops is 7, where M_{py} and M_{pz} represent respectively, the limiting bending moment about the local y'- and z'-axes of the member, and M_{px} represents the limiting torsional moment about the long axis of the member.

Loads are applied in incs increments to the nodes and the magnitude of each increment is read into the vector dload. The loading remains proportional as is customary in plastic hinge analysis, so the relative magnitudes of the nodal loads are read by node and val, where node holds the node numbers and val holds the load weightings on each freedom.

Following assembly of the global stiffness matrix and factorisation by subroutine sparin, the program enters the load increment loop. For each iteration counted by iters, the external load increments are added to the redistributive loads vector bdylds. The equilibrium equations are solved using subroutine spabac and the resulting nodal displacement increments compared with their values at the previous iteration using subroutine checon. This subroutine observes the relative change in displacement increments from one iteration to the next. If the change is less than tol then the logical variable converged is set to .TRUE and convergence has occurred. Alternatively, converged is set to .FALSE and another iteration is performed.

At each iteration, each element is inspected and its action vector computed from the product of its nodal displacements and the element stiffness matrix. The subroutine hinge adds the action vector to the values already existing from the previous load step (held in holdr) and checks both nodes to see if the plastic moment value has been exceeded. If the plastic moment value has been exceeded, the self-equilibrating vector react is formed. In Figure 4.30(a), a typical 2D element is shown in which a particular load increment has pushed the moment value at both nodes over their plastic limit. The correction vector applies a moment to each node equal to the amount of overshoot of the plastic moment values, however to preserve equilibrium, a couple is required as shown in the local coordinate system in Figure 4.30(b). Finally as shown in Figure 4.30(c), the couple is transformed into global coordinate directions before being assembled into the bdylds vector. Only those elements that have moments in excess of the plastic limits will contribute any loading to bdylds.

If, at any load step, the algorithm fails to converge within the prescribed iteration ceiling limit then "collapse" of the structure is indicated, because the algorithm has been unable to satisfy equilibrium without violating the plastic moment values.

The first example shown in Figure 4.31 is a two-bay portal frame subjected to proportional loading. After each increment, the output shown in Figure 4.32 gives the loading factor λ (equal to the accumulated values of dload) together with the loaded nodal displacements and the iteration count to achieve convergence. To reduce the volume of output, only the displacements of loaded nodes (2, 3, and 6) are given. In Figure 4.33, the horizontal movement of point A is plotted against λ, indicating close agreement with the theoretical value of $\lambda_f = 1.375$ given by Horne (1971) for this problem.

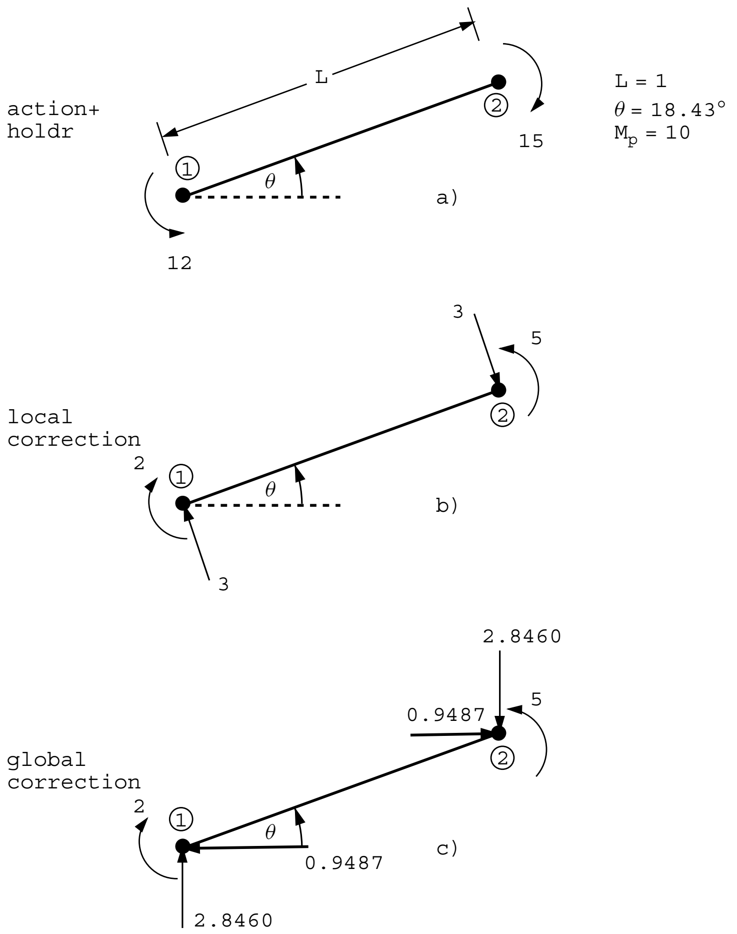


Figure 4.30 Correction terms for “yielding” element

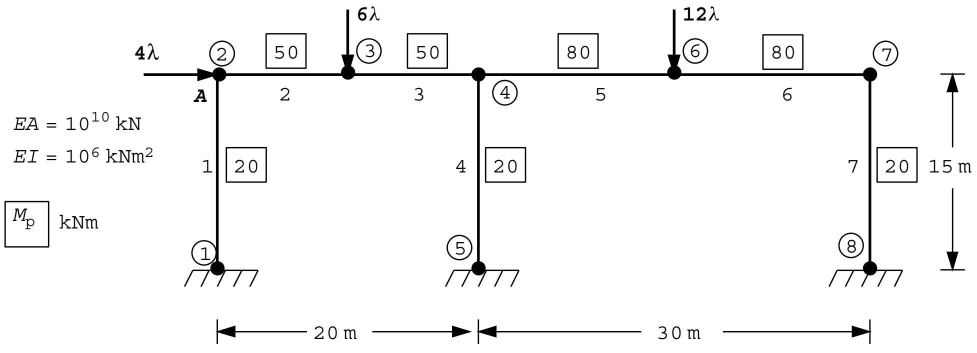


Figure 4.31 Mesh and data for first Program 4.5 example (*Continued on page 142*)

```
nels  nn  ndim  nprops  np_types
7     8    2      3       3

prop(ea,ei,mp)
1.0e10    1.0e6  20.0
1.0e10    1.0e6  50.0
1.0e10    1.0e6  80.0

etype
1  2  2  1  3  3  1

g_coord
 0.0  0.0    0.0 15.0   10.0 15.0   20.0 15.0
20.0  0.0   35.0 15.0   50.0 15.0   50.0  0.0

g_num
1 2  2 3  3 4  5 4  4 6  6 7  8 7

nr,(k,nf(:,k),i=1,nr)
3
1 0 0 0  5 0 0 0  8 0 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
3
2  4.0    0.0    0.0
3  0.0   -6.0    0.0
6  0.0  -12.0    0.0

limit  tol
200   0.0001

incs,(dload(i),i=1,incs)
8
0.5  0.3  0.2  0.2  0.1  0.05  0.02  0.01
```

Figure 4.31 (*Continued from page 141*)

```
There are   15 equations and the skyline storage is   66

Load step  1    Load factor   0.5000E+00
 Node   Displacement(s) and Rotation(s)
   2  0.2073E-03 -0.9812E-09 -0.2015E-04
   3  0.2073E-03 -0.8478E-04  0.1410E-04
   6  0.2073E-03 -0.1161E-02 -0.3057E-05
Converged in    2 iterations

Load step  2    Load factor   0.8000E+00
 Node   Displacement(s) and Rotation(s)
   2  0.4912E-03 -0.1102E-08 -0.3817E-04
   3  0.4912E-03 -0.1133E-03  0.2775E-04
   6  0.4912E-03 -0.2210E-02 -0.2088E-04
Converged in   25 iterations
.
.
.
Load step  7    Load factor   0.1370E+01
 Node   Displacement(s) and Rotation(s)
   2  0.2043E-02 -0.9284E-09 -0.1300E-03
   3  0.2043E-02 -0.2059E-03  0.9912E-04
   6  0.2043E-02 -0.5453E-02 -0.5822E-04
Converged in   65 iterations

Load step  8    Load factor   0.1380E+01
 Node   Displacement(s) and Rotation(s)
   2  0.3203E-02 -0.9416E-09 -0.1087E-03
   3  0.3203E-02  0.2461E-04  0.1241E-03
   6  0.3203E-02 -0.6727E-02 -0.5976E-04
Converged in  200 iterations
```

Figure 4.32 Results from first Program 4.5 example

The second example in 3D shown in Figure 4.34 is of a plane triangular grid, rigidly supported along one side and subjected to a point transverse load at its apex (node 7). All members have the same stiffness and plastic moment. The output shown in Figure 4.35 indicates failure occurs when the transverse load approaches a value of 1.55.

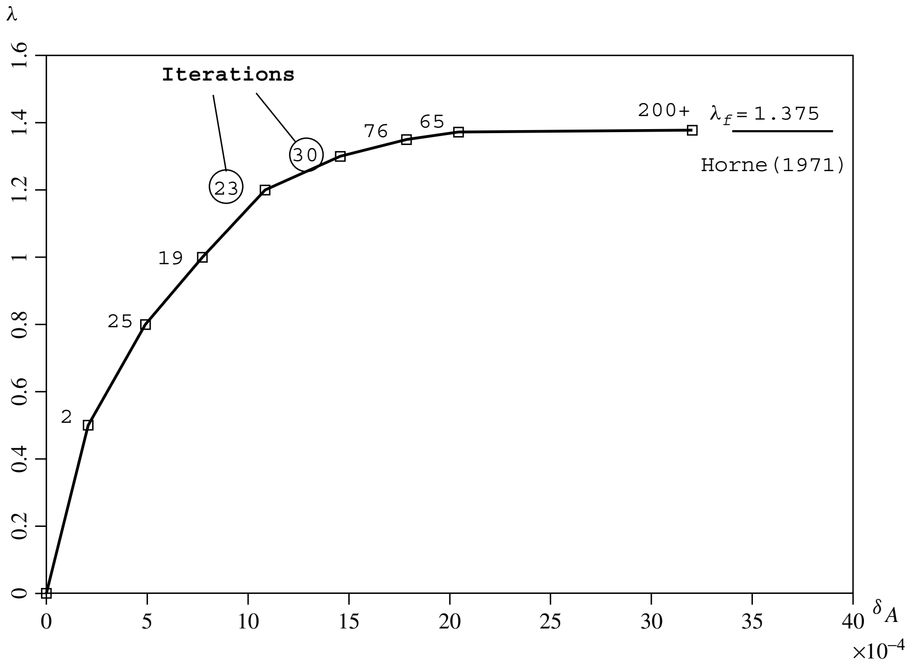


Figure 4.33 Load displacement behaviour from first Program 4.5 example

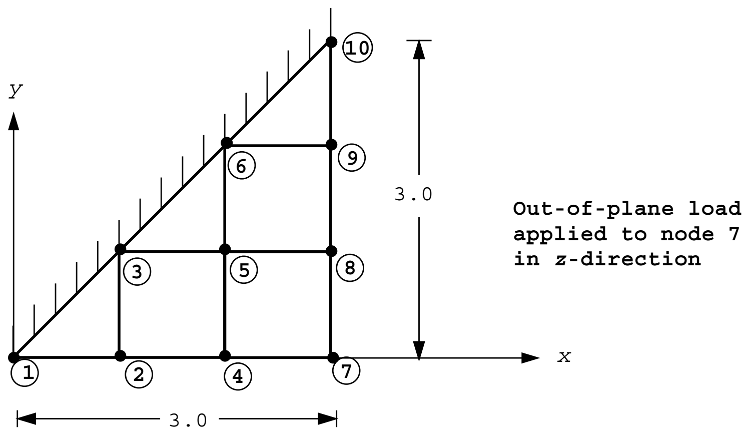


Figure 4.34 Mesh and data for second Program 4.5 example (*Continued on page 144*)

```
nels  nn  ndim  nprops  np_types
12    10   3      7       1

prop(ea,eiy,eiz,gj,mpy,mpz,mpx)
1.0  1.e4  1.e4  1.0  1.0  1.0  1.e8

etype(not needed)

gamma
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

g_coord
0.0  0.0  0.0    1.0  0.0  0.0    1.0  1.0  0.0    2.0  0.0  0.0
2.0  1.0  0.0    2.0  2.0  0.0    3.0  0.0  0.0    3.0  1.0  0.0
3.0  2.0  0.0    3.0  3.0  0.0

g_num
1 2  2 3  2 4  3 5  4 5  5 6  4 7  5 8  6 9  7 8  8 9  9 10

nr,(k,nf(:,k),i=1,nr)
4
1 0 0 0 0 0 0  3 0 0 0 0 0 0  6 0 0 0 0 0 0  10 0 0 0 0 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
1
7  0.0  0.0  1.0  0.0  0.0  0.0

limit,tol
200  0.00001

incs,(dload(i)=1,incs)
5
0.5 0.5 0.5 0.05 0.05
```

Figure 4.34 (*Continued from page 143*)

```
There are   36 equations and the skyline storage is  378
Load step  1    Load factor   0.5000E+00
 Node   Displacement(s) and Rotation(s)
   7  0.0000E+00  0.0000E+00  0.1059E-03 -0.6345E-04 -0.6345E-04  0.0000E+00
Converged in    3 iterations
Load step  2    Load factor   0.1000E+01
 Node  Displacement(s) and Rotation(s)
   7  0.0000E+00  0.0000E+00  0.2117E-03 -0.1269E-03 -0.1269E-03  0.0000E+00
Converged in    3 iterations
Load step  3    Load factor   0.1500E+01
 Node   Displacement(s) and Rotation(s)
   7  0.0000E+00  0.0000E+00  0.3247E-03 -0.1936E-03 -0.1936E-03  0.0000E+00
Converged in   18 iterations
Load step  4    Load factor   0.1550E+01
 Node   Displacement(s) and Rotation(s)
   7  0.0000E+00  0.0000E+00  0.1517E-02 -0.7944E-03 -0.7944E-03  0.0000E+00
Converged in  200 iterations
```

Figure 4.35 Results from second Program 4.5 example

Program 4.6 Stability (buckling) analysis of elastic beams using 2-node beam elements (elastic foundation optional).

```
PROGRAM p46
!-------------------------------------------------------------------------
! Program 4.6 Stability (buckling) analysis of elastic beams using 2-node
!             beam elements (elastic foundation optional).
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,iters,k,limit,ndof=4,nels,neq,nod=2,nodof=2,nn,nprops,    &
   np_types,nr
 REAL(iwp)::eval,tol,zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),kdiag(:),nf(:,:),num(:)
 REAL(iwp),ALLOCATABLE::ell(:),evec(:),kg(:,:),gv(:),km(:,:),kv(:),       &
   mm(:,:),prop(:,:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,nprops,np_types; nn=nels+1
 ALLOCATE(nf(nodof,nn),ell(nels),num(nod),g(ndof),g_g(ndof,nels),         &
   etype(nels),prop(nprops,np_types),km(ndof,ndof),kg(ndof,ndof),          &
   mm(ndof,ndof))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr),limit,tol; CALL formnf(nf)
 neq=MAXVAL(nf); ALLOCATE(kdiag(neq),evec(neq)); kdiag=0
!-----------------------loop the elements to find global array sizes------
 elements_1: DO iel=1,nels
   num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),gv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------global stiffness and geometric matrix assembly----
 kv=zero; gv=zero
 elements_2: DO iel=1, nels
   CALL beam_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
   mm=zero; IF(nprops>1)CALL beam_mm(mm,prop(2,etype(iel)),ell(iel))
   CALL beam_ge(kg,ell(iel))
   CALL fsparv(kv,km+mm,g,kdiag); CALL fsparv(gv,kg,g,kdiag)
 END DO elements_2
!-----------------------solve eigenvalue problems-------------------------
 CALL stability(kv,gv,kdiag,tol,limit,iters,evec,eval)
 WRITE(11,'(/A,E12.4,/)')" The buckling load =",eval
 WRITE(11,'(A,E12.4)')" The buckling mode =",evec(1)
 DO i=2,neq; WRITE(11,'(A,E12.4)')"                   ",evec(i); END DO
 WRITE(11,'(/A,I5,A)')" Converged in",iters," iterations"
STOP
END PROGRAM p46
```

New scalar reals:

`eval`	smallest eigenvalue (buckling load)

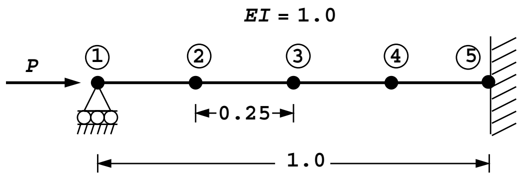
New dynamic real arrays:

`evec`	eigenvector (mode shape)
`kg`	element geometric matrix
`gv`	global geometric matrix

This program computes the fundamental (lowest) buckling load and associated mode shape of beam elements subjected to axial compressive loading. The program has much in common with Program 4.3 and includes the option of an elastic foundation. The element geometric matrices `kg` (see Section 2.5) are generated by subroutine `beam_ge` and the assembled geometric matrix stored in vector `gv`. The beam/foundation stiffness matrices are `km` and `mm`, and assembled into `kv` as in Program 4.3.

Subroutine `stability` uses simple iteration to solve the eigenvalue problem for the smallest eigenvalue called `eval`, corresponding to the lowest buckling load. The program prints the buckling load and the corresponding eigenvector `evec`, or mode shape, scaled so that its Euclidean norm equals unity. Other variables in this program relating to the eigenvalue solver include `tol`, which is the convergence tolerance for the iterative algorithm, `iters`, which holds the number of iterations to achieve convergence, and `limit` which represents the iteration ceiling.

The first example shown in Figure 4.36 is of a beam of unit length, fixed at one end and pinned at the other. Since only a beam is being analysed, `nprops=1`. Four beam elements, each of length 0.25 have been used to model the beam, and for simplicity, the flexural stiffness *EI* has also been set to unity. The output shown in Figure 4.37 gives a buckling load of 20.23, which agrees closely with the theoretical solution of $2.04\pi^2 EI/L^2 = 20.19$.

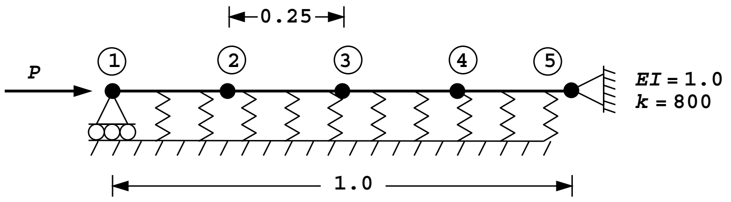


```
nels   nprops   np_types
4        1        1

prop (ei)
1.0

etype(not needed)

ell
0.25 0.25 0.25 0.25

nr,(k,nf(:,k),i=1,nr)
2
1 0 1  5 0 0

limit    tol
100      1.0e-5
```

Figure 4.36 Mesh and data for first Program 4.6 example

```
There are    7 equations and the skyline storage is   20

The buckling load = 0.2023E+02

The buckling mode = 0.7273E+00
                    0.1524E+00
                    0.3884E+00
                    0.1687E+00
                   -0.2440E+00
                    0.6725E-01
                   -0.4522E+00

Converged in   12 iterations
```

Figure 4.37 Results from first Program 4.6 example



```
nels  nprops  np_types
4       2       1

prop(ei,k)
1.0 800.0

etype(not needed)

ell
0.25 0.25 0.25 0.25

nr,(k,nf(:,k),i=1,nr)
2
1 0 1  5 0 1

limit    tol
100     1.0e-5
```

Figure 4.38 Mesh and data for second Program 4.6 example

The second example shown in Figure 4.38 is of a simply supported beam on an elastic foundation. As in the previous example, four elements have been used to discretise the problem, with the foundation stiffness set to 800.0. The result given in Figure 4.39 indicates a buckling load of 60.0 which compares well with the exact solution of 59.9 (Timoshenko and Gere, 1961). It may also be noted that the fundamental buckling mode is antisymmetric about the centreline, which is to be expected if the foundation stiffness exceeds the critical value given by $4\pi^4 EI/L^4$, which in this case equals about 390. A fuller treatment, including the case of a "follower" force is given by Smith (1979).

```
There are    8 equations and the skyline storage is   23

The buckling load =  0.6003E+02

The buckling mode =  0.5725E+00
                     0.9138E-01
                     0.1282E-05
                     0.8854E-06
                    -0.5725E+00
                    -0.9138E-01
                    -0.1282E-05
                     0.5725E+00

Converged in   26 iterations
```

Figure 4.39 Results from second Program 4.6 example

Program 4.7 Analysis of plates using 4-node rectangular plate elements. Homogeneous material with identical elements. Mesh numbered in x- or y-direction.

```
PROGRAM p47
!-------------------------------------------------------------------------
! Program 4.7 Analysis of plates using 4-node rectangular plate elements.
!             Homogeneous material with identical elements.
!             Mesh numbered in x- or y-direction.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,    &
   nip=16,nn,nod=4,nodof=4,nprops=2,np_types,nr,nxe,nye
 REAL(iwp)::aa,bb,d,d4=4.0_iwp,d12=12.0_iwp,e,one=1.0_iwp,                &
   penalty=1.0e20_iwp,th,two=2.0_iwp,v,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bm(:),coord(:,:),dtd(:,:),d2x(:),d2xy(:),d2y(:),  &
   g_coord(:,:),km(:,:),kv(:),loads(:),points(:,:),prop(:,:),x_coords(:), &
   y_coords(:),value(:),weights(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types,aa,bb,th
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),g_coord(ndim,nn),g_num(nod,nels),g(ndof),bm(3),    &
   g_g(ndof,nels),coord(nod,ndim),km(ndof,ndof),dtd(ndof,ndof),d2x(ndof), &
   d2y(ndof),d2xy(ndof),num(nod),x_coords(nxe+1),y_coords(nye+1),         &
   points(nip,ndim),weights(nip),prop(nprops,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
 DO i=1,nxe+1; x_coords(i)=(i-1)*aa; END DO
 DO i=1,nye+1; y_coords(i)=(i-1)*bb; END DO
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----------------------loop the elements to find global array sizes------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   CALL num_to_g(num,nf,g); CALL fkdiag(kdiag,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1
 CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------element stiffness integration and assembly--------
 CALL sample(element,points,weights); kv=zero
 elements_2: DO iel=1,nels
   e=prop(1,etype(iel)); v=prop(2,etype(iel))
   d=e*th**3/(d12*(one-v**2)); g=g_g(:,iel); km=zero
   integration: DO i=1,nip
     CALL fmplat(d2x,d2y,d2xy,points,aa,bb,i)
     DO k=1,ndof
         dtd(k,:)=d4*aa*bb*d*weights(i)*                                  &
         (d2x(k)*d2x(:)/(aa**4)+d2y(k)*d2y(:)/(bb**4)+(v*d2x(k)*d2y(:)+   &
         v*d2x(:)*d2y(k)+two*(one-v)*d2xy(k)*d2xy(:))/(aa**2*bb**2))
         dtd(:,k)=dtd(k,:)
     END DO
     km=km+dtd
   END DO integration
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!-----------------------read loads and/or displacements-------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)then
   ALLOCATE(node(fixed_freedoms),no(fixed_freedoms),sense(fixed_freedoms),&
     value(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 WRITE(11,'(/A)')"  Node   Disp        Rot-x       Rot-y       Twist-xy"
 DO k=1,nn; WRITE(11,'(I5,6E12.4)')k,loads(nf(:,k)); END DO
!-----------------------recover moments at element centroids--------------
 nip=1; DEALLOCATE(points); ALLOCATE(points(nip,ndim))
 CALL sample(element,points,weights)
 WRITE(11,'(/A)')(" Element x-moment    y-moment    xy-moment")
 DO iel=1,nels
   g=g_g(:,iel)
   moms: DO i=1,nip
     CALL fmplat(d2x,d2y,d2xy,points,aa,bb,i); bm=zero
     DO k=1,ndof
       bm(1)=bm(1)+d4*d*(d2x(k)/aa/aa+v*d2y(k)/bb/bb)*loads(g(k))
       bm(2)=bm(2)+d4*d*(v*d2x(k)/aa/aa+d2y(k)/bb/bb)*loads(g(k))
       bm(3)=bm(3)+d4*d*(one-v)*d2xy(k)/aa/bb*loads(g(k))
     END DO
   END DO moms
   WRITE(11,'(I5,3E12.4)')iel,bm
 END DO
STOP
END PROGRAM p47
```

New scalar integers:

`nip`	number of integration points per element
`nxe`	number of elements in the x-direction
`nye`	number of elements in the y-direction

New scalar reals:

`aa`	element dimension in x-direction
`bb`	element dimension in y-direction
`d`	plate flexural stiffness
`d4`	set to 4.0
`d12`	set to 12.0
`e`	Young's modulus
`one`	set to 1.0
`th`	plate thickness
`two`	set to 2.0
`v`	Poisson's ratio

New dynamic real arrays:

`bm`	bending moments
`dtd`	integration point contribution to `km`
`d2x`	second derivatives of shape functions with respect to ξ
`d2xy`	"mixed" second derivatives of shape functions with respect to ξ and η
`d2y`	second derivatives of shape functions with respect to η
`points`	holds sampling points in local coordinates
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout
`weights`	holds weighting coefficients for numerical integration

The previous examples have illustrated the principles of finite elements applied to "structures" made up of one-dimensional elements. Solutions to these idealised problems were not usually dependent upon the number of elements, which were chosen conveniently to reflect the positions of the load applications and changes of geometry. This example models a two-dimensional thin plate structure using a genuine finite element approximation. The number of elements used to model the plate is decided by the user, but as the number increases, so the solution should improve. The success of a finite element analysis rests on "close enough" solutions being found using a reasonable number of elements. Section 2.14 describes the governing differential equation and the finite element discretisation.

The formulation described here enforces complete compatibility of displacements between elements and equilibrium at the nodes, but there will in general be some loss of equilibrium between the nodes. Figure 4.40 illustrates a typical element and gives the freedom numbering of the `g` vector. It can be seen that there are 16 degrees of freedom per element comprising a vertical translation (w), two ordinary rotations (θ_x, θ_y) and a "twist" rotation (θ_{xy}), at each node.

The structure of the program is similar to several of the earlier programs in this chapter, except that the element stiffness matrix is calculated using (Gaussian) numerical integration in the x- and y-directions. Property data involves Young's modulus and Poisson's ratio, read into the array `prop`, the plate thickness `th` and the rectangular element dimensions `aa` and `bb` (the dimensions of the elements in the x- and y-directions respectively). The flexural stiffness of the plate `d` is calculated from the plate thickness and elastic properties. All elements are assumed to be the same size in this program and arranged into a rectangular mesh. This enables the nodal coordinates and "steering" vector for each element to be

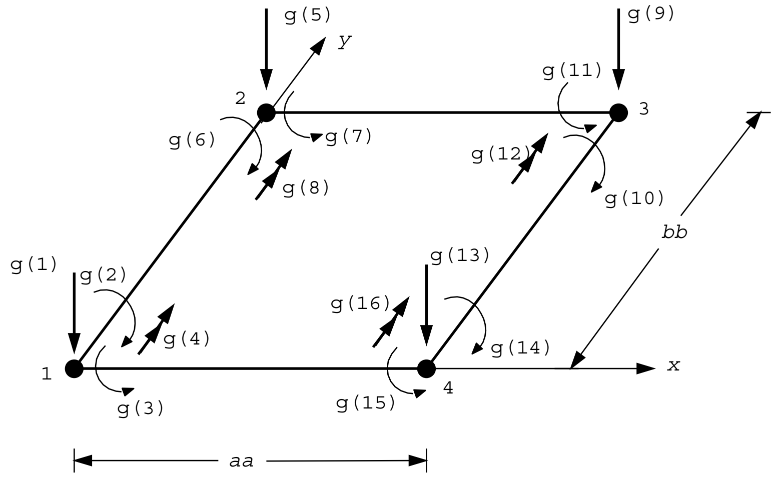


Figure 4.40 Node and freedom numbering for plate element

generated automatically by a geometry subroutine called `geom_rect`. This subroutine will be used frequently in later chapters of the book, having the ability to generate rectangular meshes for a variety of 2D elements.

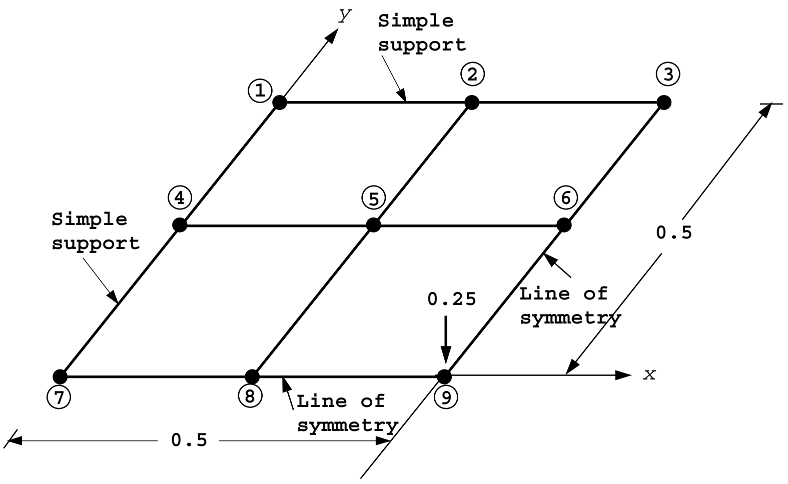
In the "element stiffness integration and assembly" loop, all the derivative arrays mentioned above are delivered for each Gauss point by the library subroutine `fmplat`. Once the Gauss point loop is completed, the element stiffness matrix is held in `km`. This is followed by assembly into a global stiffness matrix, stored as usual as a skyline vector `kv`. Nodal loads (forces, moments) and/or fixed displacements (translations, rotations) are then read and the equilibrium equations solved.

Following calculation of the global displacements, a post-processing phase begins in which the elements are scanned once more. Element nodal displacements are retrieved and the three moments (M_x, M_y and M_{xy}) are computed at the centroid of each element. It should be noted that since `nip=16` was needed for the integration phase (four Gauss points in each of the coordinate directions for exact integration), and `nip=1` is required to find the centroid of each element, it was necessary to reset `nip` to unity and "reallocate" the `points` array.

The example shown in Figure 4.41 illustrates a symmetrical quadrant of a square plate simply supported at its edges and modelled by four square elements. The plate supports a central unit load so one quarter of this value is applied to node 9.

The results in Figure 4.42 show the central deflection of the plate (node 9) to be 0.01147 which can be compared with the "exact" solution of 0.01160 (Timoshenko and Woinowsky–Krieger, 1959). By increasing the number of elements, better approximations to the exact solution are obtained.

In addition to the results file `fe95.res`, Program 4.7 is the first program in the book to output a graphics file, generically called `fe95.msh`, which holds a PostScript image of the mesh. This file is generated by subroutine `mesh`, which is one of a suite of graphics subroutines held in the library `main`. Some of the other subroutines will be described in the next chapter, and are useful for visualising results and debugging data.



```
nxe  nye  np_types
2    2    1

aa     bb    th
0.25  0.25  1.0

prop(e,v)
10.92  0.3

etype(not needed)

nr,(k,nf(:,k),i=1,nr)
8
1 0 0 0 1   2 0 0 1 1   3 0 0 1 0   4 0 1 0 1
6 1 0 1 0   7 0 1 0 0   8 1 1 0 0   9 1 0 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
1
9  0.25  0.0  0.0  0.0

fixed_freedoms
0
```

Figure 4.41 Mesh and data for Program 4.7 example

```
There are   16 equations and the skyline storage is  115
 Node    Disp         Rot-x        Rot-y        Twist-xy
    1  0.0000E+00  0.0000E+00  0.0000E+00 -0.8743E-01
    2  0.0000E+00  0.0000E+00 -0.2021E-01 -0.6814E-01
    3  0.0000E+00  0.0000E+00 -0.2956E-01  0.0000E+00
    4  0.0000E+00  0.2021E-01  0.0000E+00 -0.6814E-01
    5  0.4772E-02  0.1661E-01 -0.1661E-01 -0.6460E-01
    6  0.7125E-02  0.0000E+00 -0.2594E-01  0.0000E+00
    7  0.0000E+00  0.2956E-01  0.0000E+00  0.0000E+00
    8  0.7125E-02  0.2594E-01  0.0000E+00  0.0000E+00
    9  0.1147E-01  0.0000E+00  0.0000E+00  0.0000E+00

Element x-moment     y-moment    xy-moment
    1 -0.1193E-01 -0.1193E-01 -0.5555E-01
    2 -0.3813E-01 -0.2497E-01 -0.2804E-01
    3 -0.2497E-01 -0.3813E-01 -0.2804E-01
    4 -0.1211E+00 -0.1211E+00 -0.3347E-01
```

Figure 4.42 Results from Program 4.7 example

4.2 Concluding remarks

It has been shown how sample programs can be built up from the library of subroutines described in Chapter 3. A central feature of the programs has been their brevity. A typical main program has around 100 lines, is comprehensible to the user and is well suited for compilation on a small computer. In subsequent chapters, programs of greater complexity are introduced but the central theme of conciseness is adhered to.

Glossary of variable names used in Chapter 4

Scalar integers:

`fixed_freedoms`	number of fixed displacements
`i`	simple counter
`iel`	simple counter
`incs`	number of load increments
`iters`	counts plastic iterations
`iy`	counts load increments
`iwp`	`SELECTED_REAL_KIND(15)`
`k`	simple counter
`limit`	plastic iteration ceiling
`loaded_nodes`	number of loaded nodes
`ndim`	number of dimensions
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integration points per element
`nod`	number of nodes per element
`nodof`	number of degrees of freedom per node
`nn`	number of nodes in the mesh
`np_types`	number of different property types
`nprops`	number of material properties
`nr`	number of restrained nodes
`nxe`	number of elements in the x-direction
`nye`	number of elements in the y-direction

Scalar reals:

`aa`	element dimension in x-direction
`axial`	element axial force
`bb`	element dimension in y-direction
`d`	plate flexural stiffness
`d4`	set to 4.0
`d12`	set to 12.0
`e`	Young's modulus
`eval`	smallest eigenvalue (buckling load)
`one`	set to 1.0

`penalty`	set to 1×10^{20}
`th`	plate thickness
`tol`	plastic or eigenvalue convergence tolerance
`total_load`	running total of `dload` ($= \lambda$)
`two`	set to 2.0
`v`	Poisson's ratio
`zero`	set to 0.0

Scalar logical:

`converged`	set to `.TRUE.` if converged achieved

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`kdiag`	diagonal term location vector
`nf`	nodal freedom matrix
`no`	fixed freedom numbers vector
`node`	fixed nodes vector
`num`	element node numbers vector
`sense`	sense of freedoms to be fixed vector

Dynamic real arrays:

`action`	element nodal action vector
`bdylds`	internal correction "forces" to redistribute moments
`bm`	bending moments
`coord`	element nodal coordinates
`dload`	load increment values
`evec`	eigenvector (mode shape)
`dtd`	integration point contribution to `km`
`d2x`	second derivatives of shape functions with respect to ξ
`d2xy`	"mixed" second derivatives of shape functions with respect to ξ and η
`d2y`	second derivatives of shape functions with respect to η
`eld`	element displacement vector
`eldtot`	keeps a running total of nodal displacements
`ell`	element lengths vector
`gamma`	rotation of element about local axis
`kg`	element geometric matrix
`gv`	global geometric matrix
`g_coord`	nodal coordinates for all elements

`holdr`	holds element "actions" at convergence
`km`	element stiffness matrix
`kv`	global stiffness matrix
`loads`	global load (displacement) vector
`mm`	element "mass" matrix
`oldis`	nodal displacements from previous iteration
`points`	holds integration point (local) coordinates
`prop`	element properties matrix
`react`	element self-equilibrating "correction" vector
`val`	nodal load weightings
`value`	fixed displacements vector
`weights`	holds weighting coefficients for numerical integration
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout

4.3 Exercises

1. A simply supported beam ($L = 1$, $EI = 1$) supports a unit point transverse load ($Q = 1$) at its mid-span. The beam is also subjected to a compressive axial force P which will reduce the bending stiffness of the beam. Using two ordinary beam elements of equal length, assemble the global matrix equations for this system but do not attempt to solve them. Take full account of symmetries in the expected deformed shape of the beam to reduce the number of equations.

   Ans: $$\begin{bmatrix} (8 - P/15) & (-24 + P/10) \\ (-24 + P/10) & (96 - 12P/5) \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ -1/2 \end{Bmatrix}$$

2. Derive the mass matrix of a 3-noded 1D rod element (one node at each end and one in the middle) of unit length, cross-sectional area and density, given the following shape functions:

$$\begin{aligned} N_1 &= 2(x^2 - 1.5x + 0.5) \\ N_2 &= -4(x^2 - x) \\ N_3 &= 2(x^2 - 0.5x) \end{aligned}$$

   Ans: $$\frac{1}{30}\begin{bmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{bmatrix}$$

3. A cantilever ($L = 1$, $EI = 1$) rests on an elastic foundation of stiffness $k = 10$. A transverse point load $P = 1$ is applied at the cantilever tip. Using a single finite element, estimate the transverse deflection under the load.

   Ans: 0.188

4. The governing equation for the axial displacement u of a 1D rod embedded in an elastic medium and subjected to a distributed axial load F is:

$$EA\frac{d^2u}{dx^2} - ku + F = 0$$

where *EA* is the axial stiffness of the rod and k is the stiffness of the surrounding elastic material.

Use a 2-element discretization to estimate the axial displacement at the mid point and tip of the rod shown in Figure 4.43 which is subjected to a *point* load of 5 at its tip.

Ans: 0.0098, 0.0588

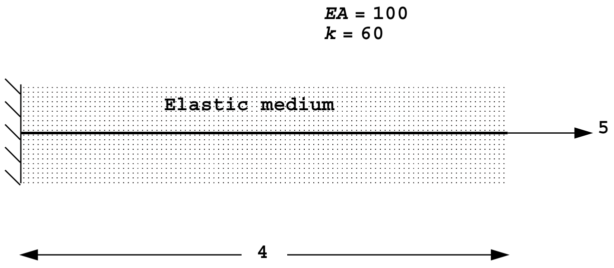


Figure 4.43

5. A propped cantilever is subjected to the loads and displacements indicated in Figure 4.44. Using two finite elements estimate the internal moment at the center of the beam.

Ans: $M = 1975$

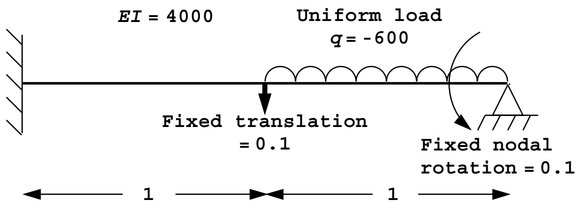


Figure 4.44

6. Use a single finite element to estimate the lowest buckling load of a column fully fixed at one end and restrained at the other in such a way that it can translate but not rotate as shown in Figure 4.45. Express your solution in terms of EI and L.

Ans: $P_{\text{crit}} = 10EI/L^2$

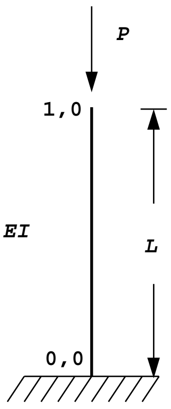


Figure 4.45

7. A cantilever of unit length and stiffness supports a unit load at its tip. The governing equation for the transverse deflection y as a function of x is given by

$$\frac{d^2y}{dx^2} = 1 - x$$

Estimate the tip deflection using the trial solution,

$$\tilde{y} = C(3x^2 - x^3)$$

and Galerkin's weighted residual method.

Ans: 1/3

8. A beam ($L = 1$, $EI = 1$), fully clamped at one end and simply supported at the other, supports a unit point transverse load ($Q = 1$) at its mid-span. The beam is also subjected to a compressive axial force P which will reduce the bending stiffness of the beam. Using two ordinary beam elements of equal length, assemble the global stiffness equations for this system in matrix form, but do not attempt to solve them.

Ans: $$\begin{bmatrix} (192 - 24P/5) & 0 & (24 - P/10) \\ 0 & (16 - 2P/15) & (4 + P/60) \\ (24 - P/10) & (4 + P/60) & (8 - P/15) \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} -1 \\ 0 \\ 0 \end{Bmatrix}$$

9. A simply supported beam of stiffness EI and length $2L$ rests on an elastic foundation of stiffness k and supports a point load P at its mid-span. By discretising half the beam, estimate the value of P that would result in a mid-span transverse deflection of 1 unit.

Ans: $P = 24EI/L^3 + 26kL/35 - (13kL^2/210 - 12EI/L^2)^2/(kL^3/105 + 4EI/L)$

10. Compute the rotation at the middle of the beam shown in Figure 4.46.

    Ans: $\theta = 0.17$

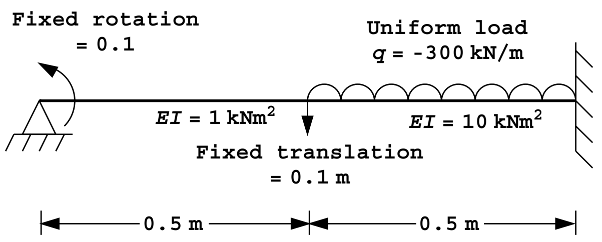


Figure 4.46

11. A beam of unit length is built in at both ends, has a stiffness $EI = 1$, and rests on an elastic foundation as shown in Figure 4.47. A unit load is applied 1/3 of the distance from one end. Estimate the value of the foundation stiffness k such that the deflection under the load is limited to 0.003.

    Ans: $k = 123$

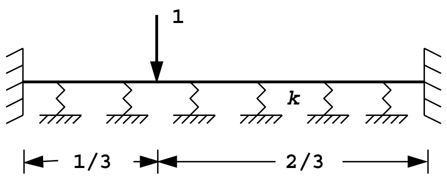


Figure 4.47

12. The continuous beam shown in Figure 4.48 is subjected to an axial load P. Using beam elements, derive the cubic expression in P which is given by the buckling loads of the system. Estimate a root of this cubic close to 5.

    Ans: $P_{\text{crit}} = 4.96$

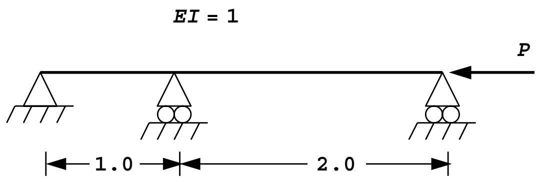


Figure 4.48

13. Use a simple finite element discretization to estimate the deflection at point A of the loaded rod shown in Figure 4.49.

    Ans: 0.075

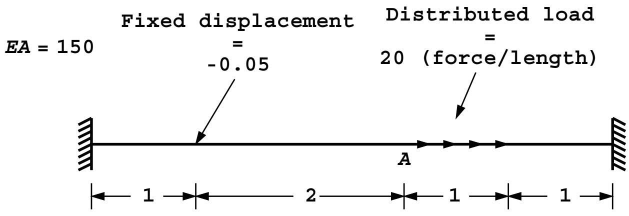

EA = 150
Fixed displacement = -0.05
Distributed load = 20 (force/length)
A
1
2
1
1

Figure 4.49

14. A beam, 20-m long, rests on the ground and is to support a uniformly distributed load of 20 kN/m. The stiffness of the ground (k) and the beam (EI) have been estimated at 10^3 kN/m^2 and 21×10^4 kNm^2 respectively. Use two finite elements with simply supported boundary conditions to estimate the central deflection of the foundation beam. (Ans: -0.022)

15. Use a finite element approach to estimate the lowest buckling load of the beam shown in Figure 4.50 which is supported along half its length by an elastic foundation. (Ans: $P_{\text{crit}} = 7.66$)

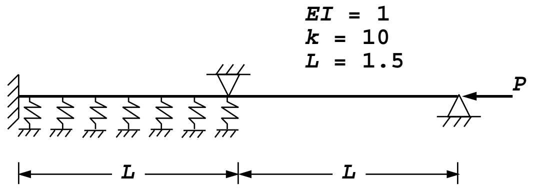


Figure 4.50

16. Use rod elements to generate the global stiffness equations due to self weight of the tower structure shown in Figure 4.51. Do not attempt to solve the equations.

Ans: $$\frac{10^8}{2}\begin{bmatrix} 4 & -4 & 0 \\ -4 & 5.44 & -1.44 \\ 0 & -1.44 & 1.60 \end{bmatrix}\begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} 274.68 \\ 373.56 \\ 109.87 \end{Bmatrix}$$

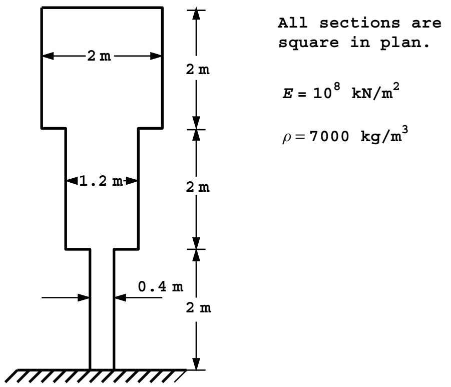


Figure 4.51

17. Use beam elements to compute the reaction force R_B and moment M_B at the right support of the uniform beam shown in Figure 4.52.

    Once you have found these values, use global equilibrium equations to compute the corresponding reactions at the left support.

    Ans: $R_A = \frac{13}{32}qL \quad M_A = \frac{11}{192}qL^2$
    $R_B = \frac{3}{32}qL \quad M_B = -\frac{5}{192}qL^2$

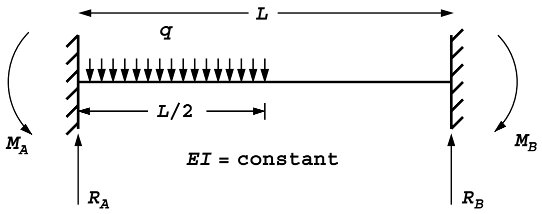


Figure 4.52

18. Compute the buckling loads of the beam shown in Figure 4.53. (Ans: $P = 735$, $P = 2099$)

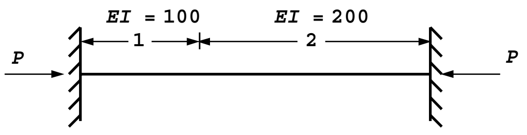


Figure 4.53

19. The structure shown in Figure 4.54, consists of three cylindrical sections with the diameters indicated. Using rod finite elements, estimate the deflection of points A and B due to self-weight, and the reaction forces at the top and bottom. (Ans: Top 573.9 kN, Bottom 30.1 kN)

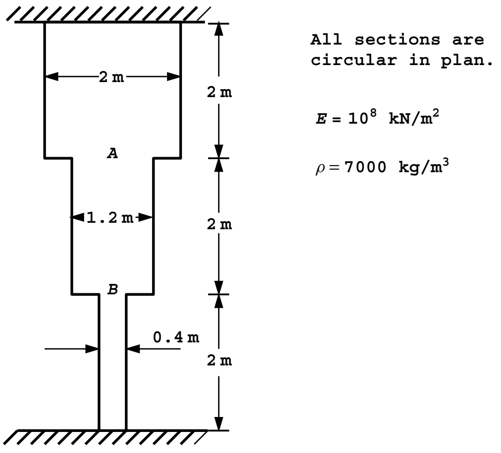


Figure 4.54

20. A framed structure is attached to a rigid table as shown in Figure 4.55. Use a finite element analysis to compute the force Q needed to push point A down to the table surface.

    Ans: $Q = 1348.5\,\text{kN}$

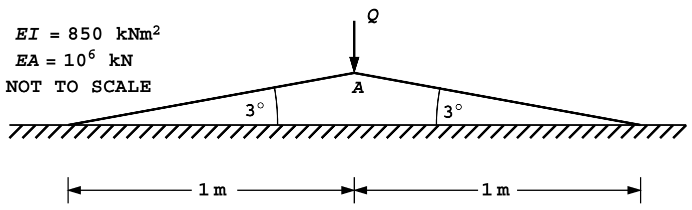


Figure 4.55

21. A simply supported beam of length L and stiffness EI supports a uniformly distributed load q and rests on an elastic foundation of stiffness k as shown in Figure 4.56. Use a single beam element to compute the end rotations, and hence estimate the mid-point translation.

    Ans: $\theta = 35qL^3/(840EI + 7kL^4)$, $w_{\text{mid}} = 35qL^4/(3360EI + 28kL^4)$

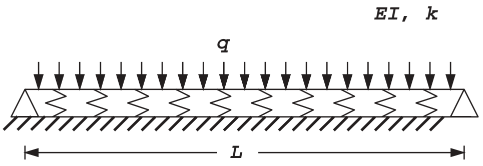


Figure 4.56

22. Compute the vertical deflection and rotation at the lower end of the system shown in Figure 4.57. Use a single beam–rod element.

    Ans: $U_1 = -0.00112$, $U_2 = 0.04792$

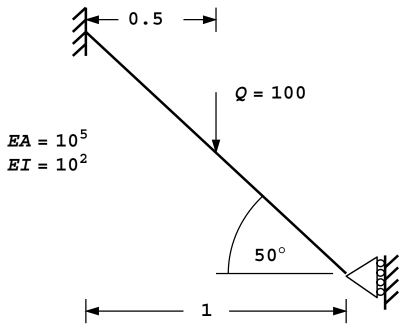


Figure 4.57

23. The propped cantilever shown in Figure 4.58 has a constant E and a linearly varying I. Use two beam elements to estimate the reaction force at B.

    Ans: $R_B = 177.1$ (analytical 168.75)

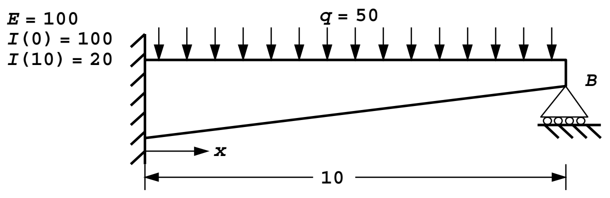


Figure 4.58

24. The column shown in Figure 4.59 has been subjected to a gradually increasing axial load P. The onset of instability was observed when $P = 102.7$. Estimate the stiffness EI of the lower half of the column.

    Ans: $EI = 100$

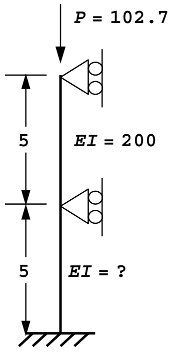


Figure 4.59

25. A single railway track resting on a ballast sub-grade can be approximated as a beam of length L of stiffness EI resting on an elastic foundation of stiffness k. If a single concentrated load P acts on a rail between the ties which can be assumed to be rigid supports, use a single finite element to estimate the relationship between EI, k, P, and L so that the rail deflection can be limited to 5 units. (Hint: Consider the cases of both simply supported and fully clamped end conditions since reality will lie somewhere in between. In the fully clamped case you will need to consider just half the problem and account for symmetry.)

    Ans: $P < \frac{320EI}{L^3} + \frac{8kL}{3}$ simply supported; $P < \frac{960EI}{L^3} + \frac{13kL}{7}$ clamped

26. A laterally loaded pile is to be modelled as a beam on an elastic foundation system as shown in Figure 4.60. Use a single beam element to estimate the lateral deflection of the pile cap under a unit load. (Hint: You may assume the base of the pile is clamped.)

    Ans: 0.489

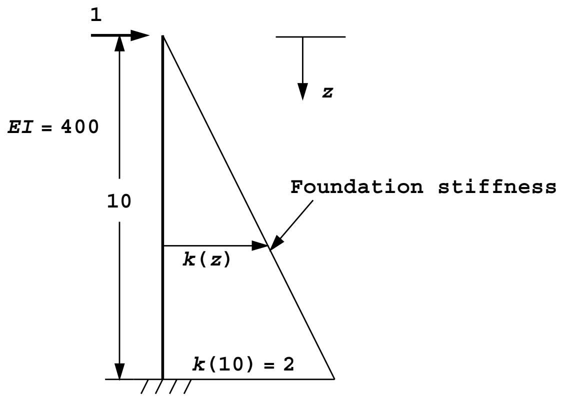


Figure 4.60

27. Use a single beam element to compute the lowest buckling load of the following cases (you may assume the flexural stiffness and length both equal unity):

    (a) A pin-ended column. Ans: 12, exact solution $= \pi^2$
    (b) A column clamped at one end and free at the other. In this case also estimate the ratio of tip rotation to translation when the column buckles. Ans: 2.486, exact solution $= \pi^2/4$; 0.638:1.000
    (c) A column clamped at both ends. Ans: 40.0, exact solution $= 4\pi^2$
    (d) A column clamped at both ends with a support preventing deflection at the mid-point. Ans: 120.0, exact solution $= 8.18\pi^2$

28. Buckling of a slender beam of stiffness *EI* resting on a uniform elastic foundation of stiffness k is governed by the equation

$$EI\frac{\partial^4 w}{\partial x^4} + P\frac{\partial^2 w}{\partial x^2} + kw = 0$$

where w is the transverse deflection of the beam and P the buckling load.

Using a single finite element, compute two buckling loads for such a beam of length L, simply supported at its ends. Show that depending on the relationship among k, *EI* and L, either of these two loads could be the critical one,

$$\text{if} \quad \theta_1 = -\theta_2, \quad P_{\text{crit}} = \frac{12EI}{L^2} + \frac{kL^2}{10}$$

$$\text{if} \quad \theta_1 = \theta_2, \quad P_{\text{crit}} = \frac{60EI}{L^2} + \frac{kL^2}{42}$$

and that the transition between these two conditions occurs when

$$k = \frac{630EI}{L^4}.$$

29. A simply supported beam element of length L, stiffness EI, resting on an elastic foundation of stiffness k supports a uniformly distributed load of q. Estimate the end rotations using a single finite element.

    Ans: $\theta = \dfrac{qL^2/12}{2EI/L + 7kL^3/420}$

30. The simply supported rigid-jointed Vierendeel girder shown in Figure 4.61 has plastic moment values of 90 kNm and 150 kNm in the vertical and horizontal members respectively. Compute the maximum point vertical load that the beam can support if it is applied at point A or point B.

    Ans: The strength of the beam is *greater* at point A (the centreline) than at point B. Use Program 4.5 to give results close to the exact solutions of $W_{ult_A} = 112$ kN and $W_{ult_B} = 99$ kN.

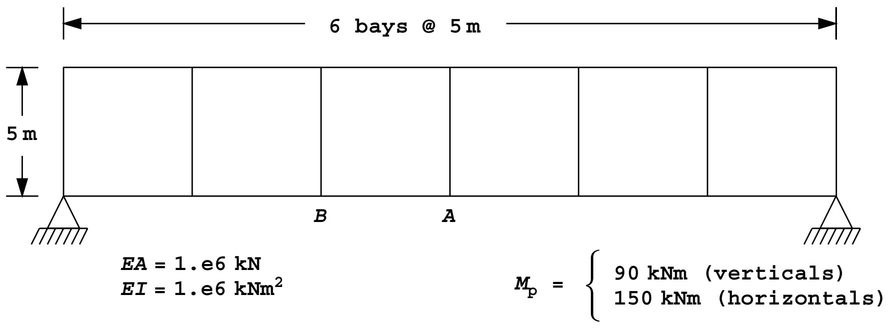


Figure 4.61

References

Griffiths DV 1988 An iterative method for plastic analysis of frames. *Comput Struct* **30**(6), 1347–1354.

Hetenyi M 1946 *Beams on Elastic Foundations*. University of Michigan Press, Ann Arbor.

Horne MR 1971 *Plastic Theory of Structures*. MIT Press, Cambridge, Mass.

Przemieniecki JS 1968 *Theory of Matrix Structural Analysis*. McGraw-Hill, New York.

Smith IM 1979 Discrete element analysis of pile instability. *Int J Numer Anal Methods Geomech* **3**, 205–211.

Timoshenko SP and Gere JM 1961 *Theory of Elastic Stability*. McGraw-Hill Book Co., New York. International Edition.

Timoshenko SP and Woinowsky-Krieger S 1959 *Theory of Plates and Shells*. McGraw-Hill, New York.

5

Static Equilibrium of Linear Elastic Solids

5.1 Introduction

This chapter describes six programs, which can be used to solve equilibrium problems in small strain solid elasticity. The programs differ only slightly from each other and, following the method adopted in Chapter 4, the first is described in some detail with changes gradually introduced into the later programs. Program 5.1 deals with 2D plane strain or axisymmetric analysis of rectangular regions using any of the 2D elements described in this book. Program 5.2 introduces 3D strain for the special case of non-axisymmetric strain of axisymmetric solids. Program 5.3 introduces conventional 3D analysis of cuboidal meshes offering a choice of hexahedral elements. Program 5.4 is a general program capable of analysing geometrically more complex problems in 2 or 3D including the use of tetrahedral elements. Program 5.5 repeats the analyses described by Program 5.3 using a mesh free pcg technique in which global stiffness assembly is avoided entirely. This procedure lends itself to vectorisation as shown in Program 5.6, which highlights some efficiency issues which arise when programming for a vector computer.

The majority of examples in this chapter consider problems involving a regular (usually rectangular or cuboidal) geometry. This has been done to simplify the presentation and minimise the volume of data required. The simple geometries enable the nodal coordinates and numbers to be generated automatically once the user has provided the element type and preferred numbering direction as data. This is done by geometry subroutines (e.g. `geom_rect` for rectangles and `hexahedron_xy` for cuboids, see Appendix E for geometry subroutine listings). For more complicated geometries, such as are possible using Program 5.4, the geometry subroutines are replaced by `read` statements for the nodal coordinates and numbering, and it is left to the user to find some other means of generating this data. Once nodal coordinates, nodal numbering, and boundary conditions are known, the next stage in all programs is to determine the element "steering vectors" `g`. These are found from `num` and `nf` as in Chapter 4, using the library subroutine `num_to_g`.

To help with debugging and to visualise results, the 2D programs in this chapter include simple graphical subroutines `mesh`, `dismsh`, and `vecmsh` which produce PostScript output files of the undeformed mesh (`fe95.msh`), the deformed mesh (`fe95.dis`), and the nodal displacement vectors (`fe95.vec`), respectively.

Program 5.1 Plane or axisymmetric strain analysis of an elastic solid using 3-, 6-, 10-, or 15-node right-angled triangles or 4-, 8-, or 9-node rectangular quadrilaterals. Mesh numbered in $x\,(r)$- or $y\,(z)$-direction.

```
PROGRAM p51
!-------------------------------------------------------------------------
! Program 5.1 Plane or axisymmetric strain analysis of an elastic solid
!             using 3-, 6-, 10- or 15-node right-angled triangles or
!             4-, 8- or 9-node rectangular quadrilaterals. Mesh numbered
!             in x(r)- or y(z)- direction.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim=2,ndof,nels,neq,nip,nn,&
   nod,nodof=2,nprops=2,np_types,nr,nst=3,nxe,nye
 REAL(iwp)::det,one=1.0_iwp,penalty=1.0e20_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::element,dir,type_2d
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),dee(:,:),der(:,:),deriv(:,:), &
   eld(:),fun(:),gc(:),g_coord(:,:),jac(:,:),km(:,:),kv(:),loads(:),      &
   points(:,:),prop(:,:),sigma(:),value(:),weights(:),x_coords(:),        &
   y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)type_2d,element,nod,dir,nxe,nye,nip,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); ndof=nod*nodof
 IF(type_2d=='axisymmetric')nst=4
 ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),fun(nod),&
   coord(nod,ndim),jac(ndim,ndim),g_num(nod,nels),der(ndim,nod),          &
   deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),eld(ndof),weights(nip),    &
   g_g(ndof,nels),prop(nprops,np_types),num(nod),x_coords(nxe+1),         &
   y_coords(nye+1),etype(nels),gc(ndim),dee(nst,nst),sigma(nst))
 READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(loads(0:neq),kdiag(neq)); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1
 CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------element stiffness integration and assembly--------
 CALL sample(element,points,weights); kv=zero; gc=one
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   int_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     IF(type_2d=='axisymmetric')THEN
       gc=MATMUL(fun,coord); bee(4,1:ndof-1:2)=fun(:)/gc(1)
     END IF
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)*gc(1)
   END DO int_pts_1
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                 &
     value(fixed_freedoms),no(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
!-----------------------equation solution--------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 IF(type_2d=='axisymmetric')THEN
   WRITE(11,'(/A)')"  Node   r-disp      z-disp"; ELSE
   WRITE(11,'(/A)')"  Node   x-disp      y-disp"
 END IF
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(nf(:,k)); END DO
!-----------------------recover stresses at nip integrating points--------
 nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
 CALL sample(element,points,weights)
 WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
 IF(type_2d=='axisymmetric')THEN
   WRITE(11,'(A,A)')" Element r-coord     z-coord",                         &
   "     sig_r       sig_z       tau_rz      sig_t"; ELSE
   WRITE(11,'(A,A)')" Element x-coord     y-coord",                         &
   "     sig_x       sig_y       tau_xy"
 END IF
 elements_3: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
   int_pts_2: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     IF(type_2d=='axisymmetric')THEN
       gc=MATMUL(fun,coord); bee(4,1:ndof-1:2)=fun(:)/gc(1)
     END IF
     sigma=MATMUL(dee,MATMUL(bee,eld)); WRITE(11,'(I5,6E12.4)')iel,gc,sigma
   END DO int_pts_2
 END DO elements_3
 CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p51
```

Scalar integers:

`fixed_freedoms`	number of fixed displacements
`i`	simple counter
`iel`	simple counter
`iwp`	`SELECTED_REAL_KIND(15)`
`k`	simple counter
`loaded_nodes`	number of loaded nodes
`ndim`	number of dimensions
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integrating points per element
`nn`	number of nodes in the mesh
`nod`	number of nodes per element
`nodof`	number of degrees of freedom per node
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes
`nst`	number of stress (strain) terms
`nxe`	number of elements in $x(r)$-direction
`nye`	number of elements in $y(z)$-direction

Scalar reals:

`det`	determinant of the Jacobian matrix
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`zero`	set to 0.0

Scalar characters:

`dir`	element and node numbering direction
`element`	element type
`type_2d`	type of 2D analysis (`'plane'` or `'axisymmetry'`)

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`kdiag`	diagonal term location vector
`nf`	nodal freedom matrix
`no`	fixed freedom numbers vector
`node`	fixed nodes vector
`num`	element node numbers vector
`sense`	sense of freedoms to be fixed vector

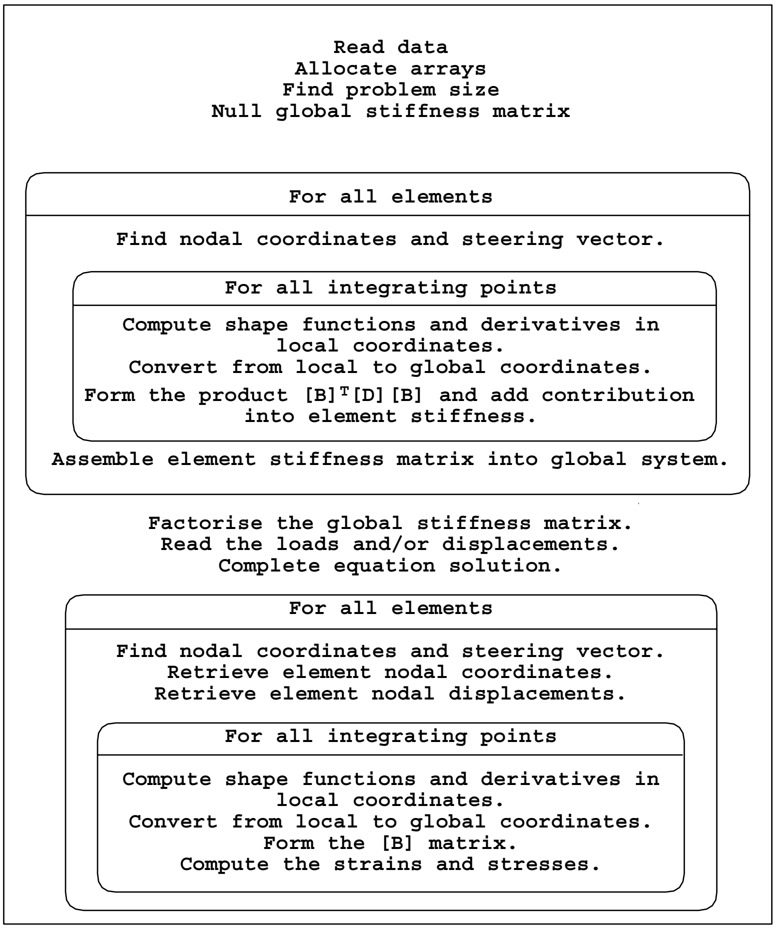
Dynamic real arrays:

`bee`	strain-displacement matrix
`coord`	element nodal coordinates
`dee`	stress strain matrix
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`eld`	element nodal displacements
`fun`	shape functions
`gc`	integrating point coordinates
`g_coord`	global nodal coordinates
`jac`	Jacobian matrix
`km`	element stiffness matrix
`kv`	global stiffness matrix
`loads`	nodal loads and displacements
`points`	integrating point local coordinates
`prop`	element properties (E and ν for each element)
`sigma`	stress terms
`value`	fixed vales of displacements
`weights`	weighting coefficients
`x_coords`	$x(r)$-coordinates of mesh layout
`y_coords`	$y(z)$-coordinates of mesh layout

The structure chart in Figure 5.1 illustrates the sequence of calculations for this program. In fact the same chart is essentially valid for all programs in this chapter which use an assembly strategy. Several different 2D elements are available for use by Program 5.1 through the input variables `element` and `nod`. Similarly, the user can select plane or axisymmetric analysis by input to `type_2d` and the numbering direction for nodes and elements by input to `dir`.

The first example for use with Program 5.1 illustrates the use of the simplest 2D element, namely the 3-noded (constant strain) triangle. This is not a very good element, and is not used much in practice, except when meshes are automatically adapted to improve accuracy (Hicks and Mar, 1996). In view of its simplicity, however, the first example in this chapter is devoted to it. As shown in the structure chart, the element stiffness matrices are formed numerically following the procedures described in Chapter 3 in equation (3.13) and Section 3.7.4. For such a simple element however, only one integrating point (`nip=1`) is required at each element's centroid.

Figure 5.2 shows a square block of elastic material of unit side length and unit thickness subjected to an equivalent vertical stress of 1 kN/m^2. The boundary conditions imply that two planes of symmetry exist and that only one quarter of the problem is being considered. The freedom numbers (non-circled) at each node represent possible displacements in the x and y directions respectively. Although this information about freedoms is included for completeness, the programs organise the information by nodes and the user need not be aware of freedom numbers at all. Figure 5.3 shows the nodal numbering system adopted for this example and although it does not matter in which direction nodes are numbered for a small problem such as this, the most efficient numbering system for general rectangular shapes will count in the direction with the least nodes. The rectangular mesh geometry



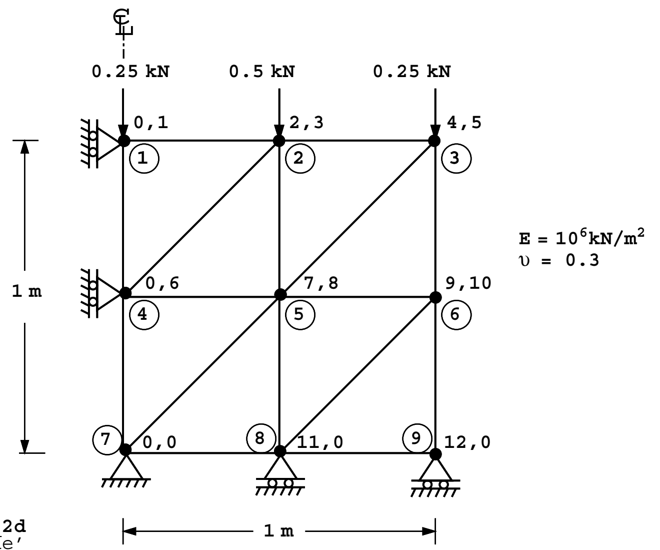
```
Read data
Allocate arrays
Find problem size
Null global stiffness matrix

For all elements
    Find nodal coordinates and steering vector.
    For all integrating points
        Compute shape functions and derivatives in
        local coordinates.
        Convert from local to global coordinates.
        Form the product [B]^T[D][B] and add contribution
        into element stiffness.
    Assemble element stiffness matrix into global system.

Factorise the global stiffness matrix.
Read the loads and/or displacements.
Complete equation solution.

For all elements
    Find nodal coordinates and steering vector.
    Retrieve element nodal coordinates.
    Retrieve element nodal displacements.
    For all integrating points
        Compute shape functions and derivatives in
        local coordinates.
        Convert from local to global coordinates.
        Form the [B] matrix.
        Compute the strains and stresses.
```

Figure 5.1 Structure chart for all Chapter 5 programs involving assembly

generated by subroutine `geom_rect` assumes that all elements are right-angled, congruent and formed by diagonal lines drawn from the bottom left corner to the top right corner of the surrounding rectangles. Figure 5.3 shows the node and element numbering for this case assuming `dir = 'x'`. Figure 5.4 shows the order of node and freedom numbering at the element level. Node 1 can be any corner, but subsequent corners and freedoms must follow in a clockwise sense. Thus, the top left element (`iel=1`) in Figure 5.2 has a steering vector $\texttt{g} = [\,0\;1\;2\;3\;0\;6\,]^T$ and its neighbour (`iel=2`) has a steering vector $\texttt{g} = [\,7\;8\;0\;6\;2\;3\,]^T$.

It is expected that, where necessary, users will replace the geometry subroutine `geom_rect` by more sophisticated versions. It need only be ensured that the coordinates and node numbers are generated consistently.

Referring to Figure 5.2, the first line of data reads the type of strain conditions `type_2d='plane'`, thus a plane-strain analysis is to be performed. The next line reads `element='triangle'`, `nod=3`, and `dir='x'`, which indicates that 3-noded triangles will be used with node and element numbering in the x-direction. The next line reads `nxe=2`,



```
type_2d
'plane'

element      nod  dir
'triangle'    3   'x'

nxe  nye  nip  np_types
2    2    1    1

prop(e,v)
1.0e6  0.3

etype(not needed)

x_coords, y_coords
0.0  0.5  1.0
0.0 -0.5 -1.0

nr,(k,nf(:,k),i=1,nr)
5
1 0 1  4 0 1  7 0 0  8 1 0  9 1 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
3
1  0.0 -0.25   2  0.0 -0.50   3  0.0 -0.25

fixed_freedoms
0
```

Figure 5.2 Mesh and data for first Program 5.1 example

nye=2, which sets the mesh to consist of two columns and two rows of elements respectively, with the diagonals forming triangles as referred to above, nip=1 which sets the numerical integration to use one integrating point per element and np_types which indicates that there is only one property group in this homogeneous example. As usual, since np_types equals 1, the etype data is not required. The next line reads the properties which in an elastic analysis consist of Young's modulus and Poisson's ratio, set respectively to 1×10^6 kN/m^2 and 0.3. The next two lines read the x-coordinates (x_coords) and y-coordinates (y_coords) of the vertical and horizontal lines that make up the mesh. Nodal freedom data concerning boundary restraints is read next, consisting of the number of restrained nodes, nr=5, followed by the restrained node number and a binary "on-off" switch corresponding to the x- and y-displacement components. For example 1 0 1 means

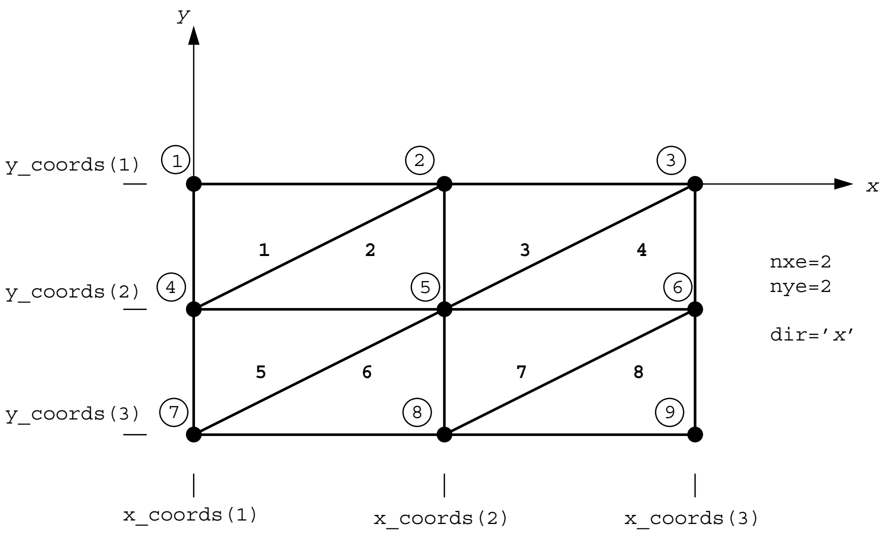


Figure 5.3 Global node and element numbering for mean of 3-node triangles

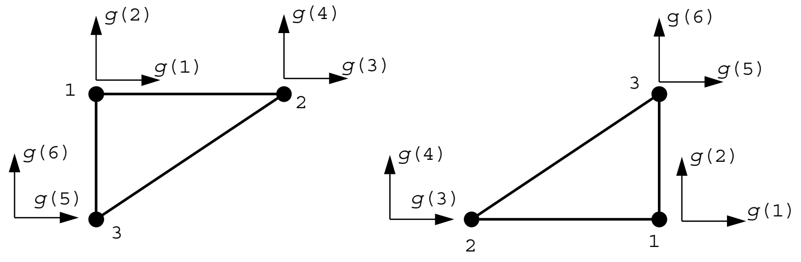


Figure 5.4 Local node and freedom numbering for different orientations of 3-node triangles

that at node 1, the x-displacement is equal to zero while the $y-$displacement remains free, and 7 0 0 means that node 7 is completely restrained. The final part of the data file refers to loads and fixed displacement data. In this example, a uniform pressure of 1 kN/m^2 is to be applied to the top surface of the block, which in the data file is replaced by equivalent nodal loads. In the case of the 3-node triangle, the total force on each element is simply shared equally between the two nodes (see Appendix A). In this case, loaded_nodes is read as 3, representing the nodes at the top of the block, and this is followed by the node number and the x- and y-components of load to be applied. There are no fixed non-zero displacements in this example, so fixed_freedoms is read as zero.

After declaration of arrays whose dimensions are known, the program enters the "input and initialisation" stage. Data concerning the mesh and its properties are now presented together with the nodal freedom data as given in Figure 5.2. The total number of nodes nn and equations in the problem neq, are provided by subroutine mesh_size.

In the section called "loop the elements to find global arrays sizes", the elements are looped to generate "global" arrays containing the element node numbers (g_num), the element nodal coordinates (g_coord), and the element steering vectors (g_g). Also within this loop, the array kdiag is formed, which holds the addresses of the skyline storage leading diagonal terms (see Figure 3.18). In larger problems, a bandwidth optimiser (Cuthill and McKee, 1969) will improve efficiency by re-ordering the nodes. Immediately following this loop, the subroutine mesh generates a Postscript file of the mesh held in file fe95.msh.

The section called "element stiffness integration and assembly" is now entered, and begins with a call to subroutine sample, which forms the quadrature sampling points and weights. The elements are then looped once more, and the nodal coordinates coord and the steering vector g for each element are retrieved.

After the stiffness matrix km has been nulled, the integration loop is entered. The local coordinates of each integrating point (only 1 in this case) are extracted from points, and the derivatives of the shape functions with respect to those coordinates der are provided for the 3-node element by the library subroutine shape_der. The conversion of these derivatives to the global system deriv requires a sequence of subroutine calls described by equations (3.47) to (3.48). The bee matrix is then formed by the subroutine beemat.

The next line adjusts the bee matrix for axisymmetry if needed and then the contribution from each integration point from (3.51) is scaled by the weighting factor from weights and added into the element stiffness matrix km. Eventually, the completed km is assembled into the global stiffness kv using the library subroutine fsparv which makes use of kdiag, as was used extensively in Chapter 4.

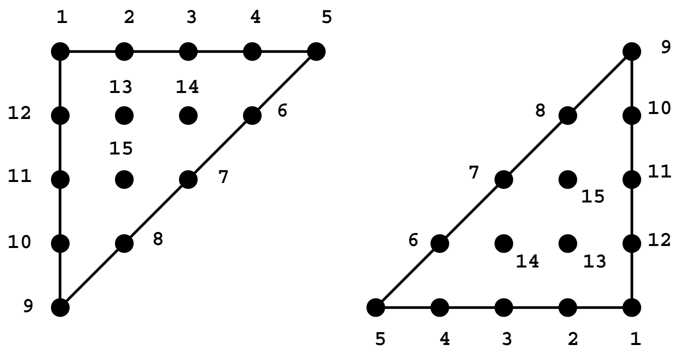
When all element stiffnesses have been assembled, the program enters the "equation solution" stage. The loads and/or fixed displacements are read, and in the case of fixed displacements, the stiffness matrix kv is modified using the "stiff spring" or "penalty" technique (Section 3.6) encountered previously in Chapter 4. The equation solution is in two stages; first the global matrix kv is factorised by subroutine sparin and this is followed by the forward- and back-substitution stage by subroutine spabac. The solution vector holding the nodal displacements (still called loads) is printed.

If required, the strains and stresses within the elements can now be computed in the section called "recover stresses at nip integrating points". These could be found anywhere in the elements by computing the bee matrix at the required locations, but it is convenient and often more accurate to employ the integrating points that were used in the stiffness formulation. In this example only one integrating point at the element centroid was employed for each element, so it is at this location that strains and stresses will be calculated. Each element is scanned once more and its nodal displacements eld retrieved from the global displacement vector loads. The bee matrix for each integrating point is recalculated and the product of bee and eld yields the strains from equation (3.52). Multiplication by the stress–strain matrix dee gives the stresses sigma from equation (3.54) which are printed.

The computed results for the example shown in Figure 5.2 are given in Figure 5.5. For this simple case the results are seen to be "exact". The vertical displacements under the loads (nodes 1, 2, and 3) all equal 0.91×10^{-6} m and the Poisson's ratio effect has caused horizontal movement at nodes 3, 6, and 9 to equal 0.39×10^{-6} m. The stress components give the expected equilibrium values of $\sigma_y = -1.0$ and $\sigma_x = \tau_{xy} = 0$.

```
There are   12 equations and the skyline storage is   54

 Node   x-disp      y-disp
   1  0.0000E+00 -0.9100E-06
   2  0.1950E-06 -0.9100E-06
   3  0.3900E-06 -0.9100E-06
   4  0.0000E+00 -0.4550E-06
   5  0.1950E-06 -0.4550E-06
   6  0.3900E-06 -0.4550E-06
   7  0.0000E+00  0.0000E+00
   8  0.1950E-06  0.0000E+00
   9  0.3900E-06  0.0000E+00

The integration point (nip= 1) stresses are:
Element x-coord     y-coord     sig_x       sig_y       tau_xy
   1  0.1667E+00 -0.1667E+00  0.0000E+00 -0.1000E+01 -0.8145E-16
   2  0.3333E+00 -0.3333E+00  0.3331E-15 -0.1000E+01 -0.1629E-15
   3  0.6667E+00 -0.1667E+00  0.1110E-15 -0.1000E+01  0.3665E-15
   4  0.8333E+00 -0.3333E+00  0.4441E-15 -0.1000E+01  0.1629E-15
   5  0.1667E+00 -0.6667E+00  0.2220E-15 -0.1000E+01 -0.1222E-15
   6  0.3333E+00 -0.8333E+00  0.5551E-15 -0.1000E+01 -0.1425E-15
   7  0.6667E+00 -0.6667E+00  0.0000E+00 -0.1000E+01  0.1833E-15
   8  0.8333E+00 -0.8333E+00  0.2220E-15 -0.1000E+01 -0.4072E-16
```

Figure 5.5 Results from first Program 5.1 example

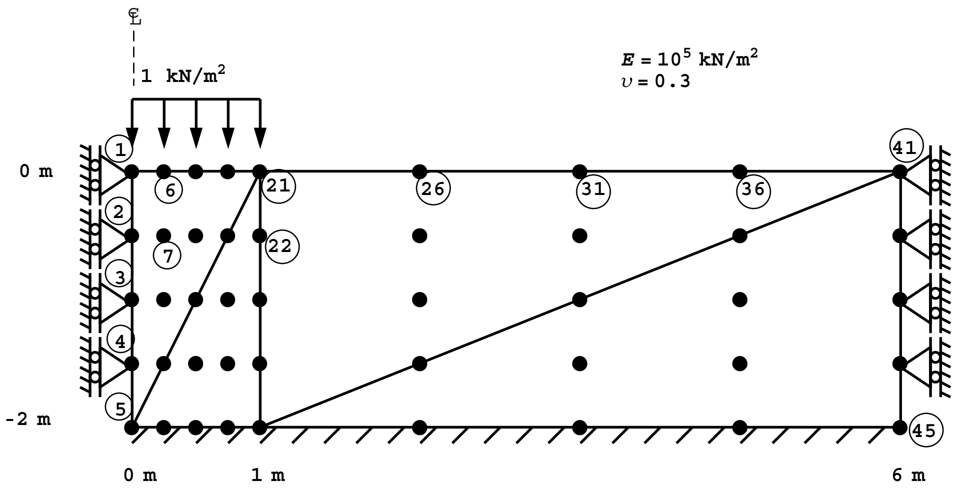


Freedoms numbered from 1 to 30 in same order as the nodes

Figure 5.6 Local node and freedom numbering for different orientations of 15-node triangles

Program 5.1 is able to use both 6-node and 10-node triangular elements, but the next member of the triangular element family to be considered is the 15-node “cubic strain” triangle (see Appendix B). The node numbering system for all triangles involves starting at a corner and progressing clockwise. Internal nodes, if present, (e.g. 10- and 15-noded triangles) are numbered last. The node numbering at the element level for a 15-noded triangle is shown in Figure 5.6. It is seen that the three internal nodes are also numbered in a clockwise sense.

The second example and data in Figure 5.7 show half of a flexible footing resting on a uniform elastic layer supporting a uniform pressure of 1 kN/m^2. Because of symmetry, only half of the layer needs to be analysed and the width has been arbitrarily terminated at



```
type_2d
'plane'

element      nod  dir
'triangle'    15  'y'

nxe  nye  nip  np_types
2     1    12    1

prop(e,v)
1.0e5  0.2

etype(not needed)

x_coords, y_coords
0.0   1.0  6.0
0.0  -2.0

nr,(k,nf(:,k),i=1,nr)
17
 1 0 1   2 0 1   3 0 1   4 0 1   5 0 0
10 0 0  15 0 0  20 0 0  25 0 0  30 0 0
35 0 0  40 0 0  41 0 1  42 0 1  43 0 1
44 0 1  45 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
5
 1  0.0 -0.0778    6  0.0 -0.3556   11  0.0 -0.1333
16  0.0 -0.3556   21  0.0 -0.0778

fixed_freedoms
0
```

Figure 5.7 Mesh and data for second Program 5.1 example

a roller boundary at six times the load width from the centreline. The data indicate that a 15-noded triangle is to be used in a plane strain analysis, with node and element numbering in the y-direction. The relatively high order of the interpolating polynomials associated with this element, suggests that fewer elements would be required for a typical boundary value problem than if working with a lower order element. The mesh shown in Figure 5.7 consists of two columns of elements (`nxe=2`) and one row of elements (`nye=1`). The recommended

```
There are   64 equations and the skyline storage is 1050

 Node   x-disp      y-disp
   1  0.0000E+00 -0.1591E-04
   2  0.0000E+00 -0.1158E-04
   3  0.0000E+00 -0.7226E-05
   4  0.0000E+00 -0.3354E-05
   5  0.0000E+00  0.0000E+00
   6 -0.9321E-06 -0.1559E-04
   7  0.1493E-06 -0.1128E-04
   8  0.4540E-06 -0.7019E-05
   9  0.3347E-06 -0.3255E-05
  10  0.0000E+00  0.0000E+00
.
.
.
  44  0.0000E+00 -0.1906E-07
  45  0.0000E+00  0.0000E+00

The integration point (nip= 1) stresses are:
Element x-coord     y-coord     sig_x       sig_y       tau_xy
   1  0.3333E+00 -0.6667E+00 -0.8302E-01 -0.9098E+00  0.7671E-01
   2  0.6667E+00 -0.1333E+01 -0.4434E-01 -0.6555E+00  0.1123E+00
   3  0.2667E+01 -0.6667E+00 -0.2042E-01  0.3240E-01 -0.1323E-01
   4  0.4333E+01 -0.1333E+01 -0.7382E-02  0.1345E-01 -0.3256E-02
```

Figure 5.8 Results from second Program 5.1 example

number of integrating points for this element in plane strain is `nip=12`. The data follow a similar pattern to the previous example. In this case, the equivalent nodal loads for a 15-noded triangle are not intuitive, and Appendix A gives the required values to reproduce a unit pressure under the “footing”.

The computed results for this example, given in Figure 5.8, indicate a centreline displacement of -0.1591×10^{-4} m. This is in good agreement with the solution of -0.153×10^{-4} m given by Poulos and Davis (1974). In order to minimise the volume of output, Program 5.1 always computes and prints stresses at element centroids. This is easily achieved in the main program by redefining `nip=1`, followed by a reallocation of the `points` and `weights` arrays (having first been “deallocated”). Users are of course free to remove these lines, and print the stresses at other locations if required.

The third example demonstrates the 4-node “linear strain” quadrilateral. Figure 5.9 shows a typical mesh of elements, together with the node and element numbering in the case of numbering in the y-direction (`dir='y'`). Figure 5.10 gives the node numbering system adopted for the 4-node quadrilateral and also the order in which the recommended number of integrating points `nip=4` are visited. Consistent with triangular elements, nodal numbering always starts at a corner and proceeds clockwise.

Figure 5.11 shows the mesh and data for a rigid strip footing resting on a uniform elastic layer. In this case the footing is given a fixed displacement in the y-direction of -1×10^{-5} m at nodes 1 and 4 into the layer. Since there are no applied loads, `loaded_nodes` is read as zero. The two fixed displacements are entered by reading `fixed_freedoms` as 2, followed by, for each fixed freedom, the node to be fixed (1 and 4), the sense of the fixity (2), and the magnitude of the fixed displacement (-1×10^{-5} m).

The computed results in Figure 5.12 confirm the fixed y-displacements at nodes 1 and 4 have the expected value of -1×10^{-5} m. Node 7 has moved up by 0.1258×10^{-5} m,

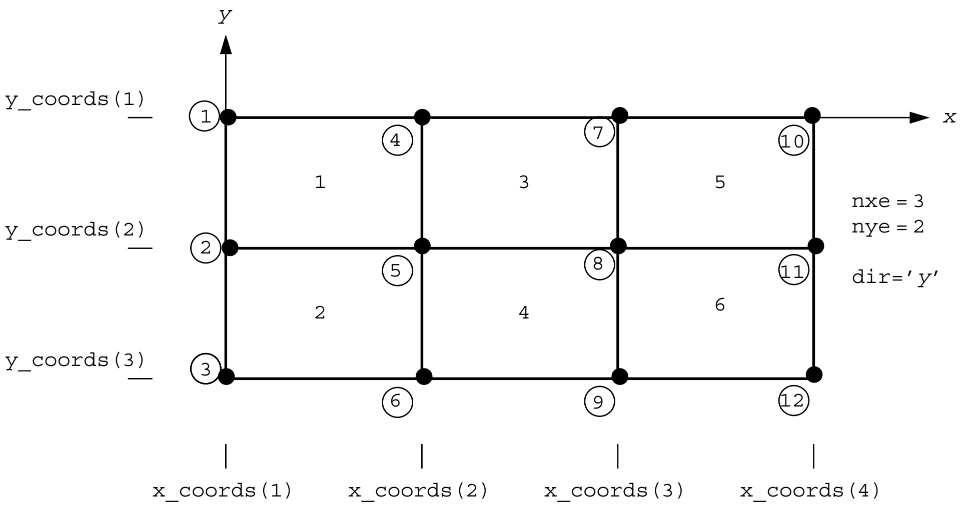


Figure 5.9 Global node and element numbering for mesh of 4-node quadrilaterals numbered in the 'y' direction

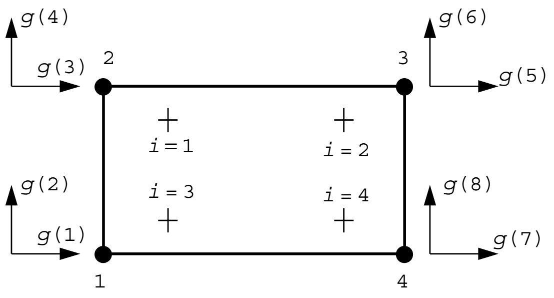
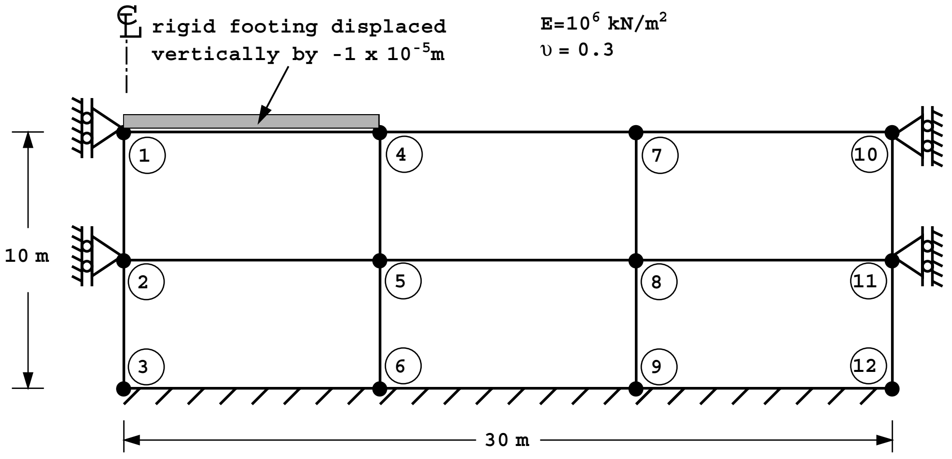


Figure 5.10 Local node, freedom and Gauss point numbering for the 4-node quadrilateral (nip = 4)

and the vertical stress at the centroid of the element immediately beneath the load gives $\sigma_y = -1.332\,\text{kN/m}^2$. Comparison with closed form or other numerical solutions will show that, with such a coarse mesh of these elements, these results can be quite inaccurate. Such discretisation errors are inevitable in finite element work, and it is the user's responsibility to experiment with mesh designs to help discover whether the numerical solution is adequate.

The fourth example, shown in Figure 5.13, illustrates the use of a higher-order element, namely the 8-noded quadrilateral, with nodes numbered in the x-direction. The local node and freedom numbering for this element as shown in Figure 5.14 indicate as usual, that node 1 is assigned to a corner and the rest follow in a clockwise sense. The general 8-node quadrilateral element stiffness matrix contains fourth order polynomial terms and thus requires `nip` to be 9 for "exact" integration. It is often the case, however, that the



```
type_2d
'plane'

element          nod  dir
'quadrilateral'   4   'y'

nxe  nye  nip  np_types
3     2    4     1

prop(e,v)
1.0e6  0.3

etype(not needed)

x_coords, y_coords
0.0  10.0  20.0  30.0
0.0  -5.0 -10.0

nr,(k,nf(:,k),i=1,nr)
8
1 0 1  2 0 1  3 0 0  6 0 0  9 0 0  10 0 1  11 0 1  12 0 0

loaded_nodes
0

fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
2
1  2  -1.0e-5   4  2  -1.0e-5
```

Figure 5.11 Mesh and data for third Program 5.1 example

use of “reduced” integration, by putting `nip` equal to 4, improves the performance of this element. This is found to be particularly true of the plasticity applications described in Chapter 6.

The simple mesh in Figure 5.15 is to be analysed and the consistent nodal loads (Appendix A) necessary to reproduce a uniform stress field of 1 kN/m^2 should be noted in the data. The computed results given in Figure 5.16, indicate a vertical displacement at node 1 of -0.5311×10^{-5} m and a vertical centroid stress in the element under the load of $\sigma_y = -0.9003$ kN/m^2.

```
There are   12 equations and the skyline storage is   58

 Node   x-disp      y-disp
   1  0.0000E+00 -0.1000E-04
   2  0.0000E+00 -0.5152E-05
   3  0.0000E+00  0.0000E+00
   4  0.8101E-07 -0.1000E-04
   5  0.1582E-05 -0.4594E-05
   6  0.0000E+00  0.0000E+00
   7  0.1241E-06  0.1258E-05
   8  0.1472E-05  0.1953E-06
   9  0.0000E+00  0.0000E+00
  10  0.0000E+00  0.2815E-06
  11  0.0000E+00  0.3475E-06
  12  0.0000E+00  0.0000E+00

The integration point (nip= 1) stresses are:
Element x-coord     y-coord     sig_x       sig_y       tau_xy
   1  0.5000E+01 -0.2500E+01 -0.4796E+00 -0.1332E+01 -0.4699E-01
   2  0.5000E+01 -0.7500E+01 -0.4558E+00 -0.1266E+01  0.7160E-01
   3  0.1500E+02 -0.2500E+01 -0.2551E+00 -0.5867E+00  0.1990E+00
   4  0.1500E+02 -0.7500E+01 -0.2611E+00 -0.5952E+00  0.2096E+00
   5  0.2500E+02 -0.2500E+01 -0.4995E-01  0.8810E-01 -0.6770E-01
   6  0.2500E+02 -0.7500E+01 -0.6777E-01  0.3061E-01  0.5955E-01
```

Figure 5.12 Results from third Program 5.1 example

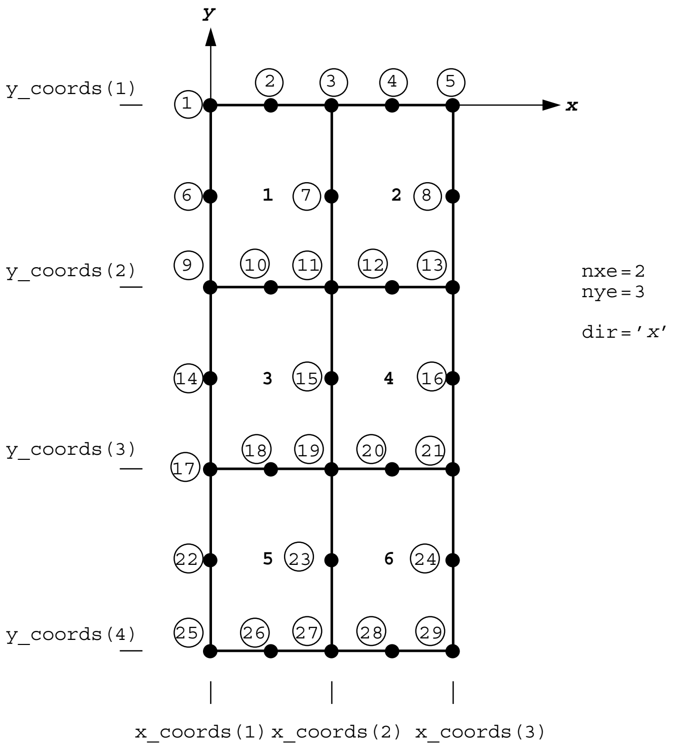


Figure 5.13 Global node and element numbering for mesh of 8-node quadrilaterals numbered in the 'x' direction

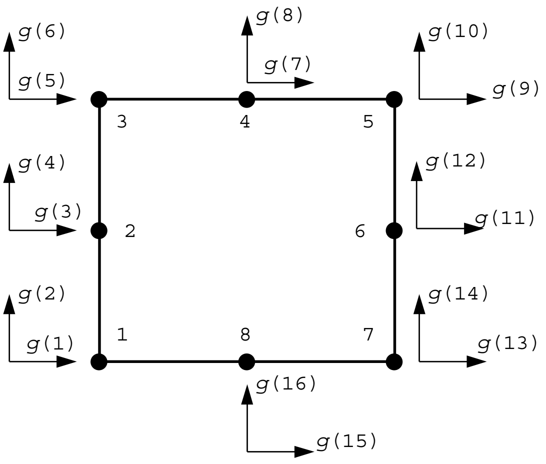
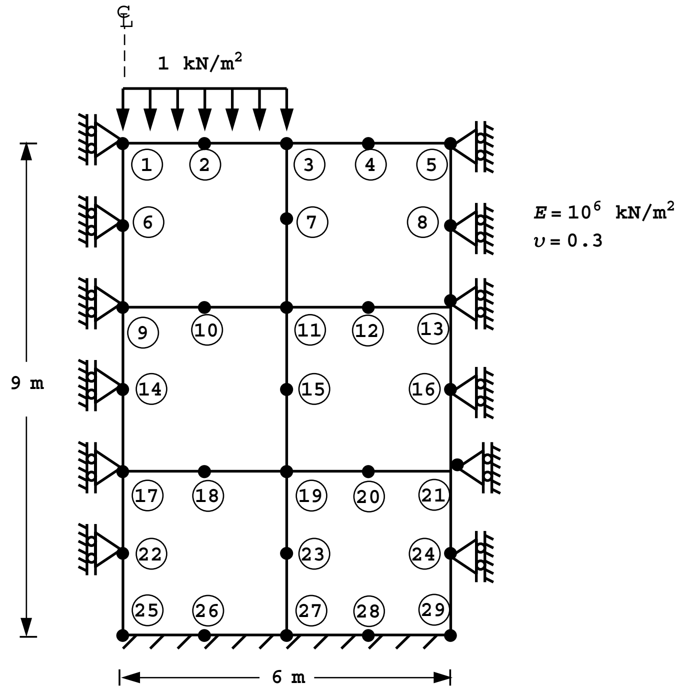


Figure 5.14 Local node and freedom numbering for the 8-node quadrilateral

The fifth example and data shown in Figure 5.17 illustrates an axisymmetric foundation analysis (`type_2d='axisymmetric'`) as opposed to the plane strain analyses used in the previous examples. The mesh, while still "rectangular", involves 4-node quadrilateral elements of variable size. Node and element numbering is in the "depth" or `'z'` direction. The mesh size data `nxe` and `nye` in an axisymmetric context, should be interpreted as the number of "columns" in the radial direction and the number of rows in the depth direction respectively. Axisymmetric integration is never "exact" using conventional Gaussian quadrature in elastic analysis, due to the $1/r$ terms that appear in the integrand of the element stiffness matrix. This example uses `nip=9`, but slightly different results can be expected as `nip` is increased. This example introduces variable properties in which E and ν are allowed to assume different values in each horizontal layer of elements. In this case there are two property groups, so `np_types` is read as 2, and two lots of properties are read into the array `prop`. Since `np_types` is greater than 1, then `etype` data is needed and takes the form of integers 1 or 2 for each element, remembering that the mesh elements are numbered in the `'z'` direction.

It should be noted that in axisymmetry, four components of strain and stress are required, so the main program sets `nst` to 4, and the appropriate `dee` matrix (2.77) is returned by subroutine `deemat`. Furthermore, axisymmetric conditions require the `bee` matrix to have a fourth row (2.76), and integration (2.74) involves the radius of each integrating point held in `gc(1)`. The main program checks whether `type_2d` is equal to `'axisymmetry'` and makes these adjustments as necessary.

The nodal loads imply a uniform stress of 1 kN/m^2 is to be applied to a one radian area of radius 10 m (see Appendix A). The computed results for this problem, including stresses at the element centroids, are given in Figure 5.18. Thus the centreline z-displacement is computed to be -0.3176×10^{-1} m and the vertical central stress in the depth direction within element 1 is $\sigma_z = -1.073$ kN/m^2.



```
type_2d
'plane'

element             no d  dir
'quadrilateral'      8     'x'

nxe  nye  nip  np_t ypes
2     3    4     1

prop(e,v)
1.0e6  0.3

etype(not needed)

x_coords, y_coords
0.0  3.0  6.0
0.0 -3.0 -6.0 -9.0

nr,(k,nf(:,k),i=1,n r)
17
 1 0 1   6 0 1   9   0 1  14 0 1  17 0 1   22 0 1  25 0 0
26 0 0  27 0 0  28   0 0   5 0 1   8 0 1   13 0 1  16 0 1
21 0 1  24 0 1  29   0 0

loaded_nodes,(k,loa ds(nf(:,k)),i=1,loa ded_nodes)
3
1  0.0 -0.5   2  0. 0 -2.0   3  0.0 -0. 5

fixed_freedoms
0
```

Figure 5.15 Mesh and data for fourth Program 5.1 example

```
 There are   36 equations and the skyline storage is  390

  Node   x-disp      y-disp
    1  0.0000E+00 -0.5311E-05
    2 -0.4211E-06 -0.5041E-05
    3 -0.7222E-06 -0.3343E-05
    4 -0.4211E-06 -0.1644E-05
    5  0.0000E+00 -0.1375E-05
    6  0.0000E+00 -0.4288E-05
    7  0.3774E-06 -0.2786E-05
    8  0.0000E+00 -0.1283E-05
    9  0.0000E+00 -0.3243E-05
   10  0.2708E-06 -0.2873E-05
.
.
.
   25  0.0000E+00  0.0000E+00
   26  0.0000E+00  0.0000E+00
   27  0.0000E+00  0.0000E+00
   28  0.0000E+00  0.0000E+00
   29  0.0000E+00  0.0000E+00

 The integration point (nip= 1) stresses are:
 Element x-coord     y-coord     sig_x       sig_y       tau_xy
    1  0.1500E+01 -0.1500E+01 -0.2476E+00 -0.9003E+00  0.1040E+00
    2  0.4500E+01 -0.1500E+01 -0.1810E+00 -0.9973E-01  0.1040E+00
    3  0.1500E+01 -0.4500E+01 -0.1683E+00 -0.6489E+00  0.8714E-01
    4  0.4500E+01 -0.4500E+01 -0.2602E+00 -0.3511E+00  0.8714E-01
    5  0.1500E+01 -0.7500E+01 -0.1994E+00 -0.5612E+00  0.2888E-01
    6  0.4500E+01 -0.7500E+01 -0.2292E+00 -0.4388E+00  0.2888E-01
```

Figure 5.16 Results from fourth Program 5.1 example

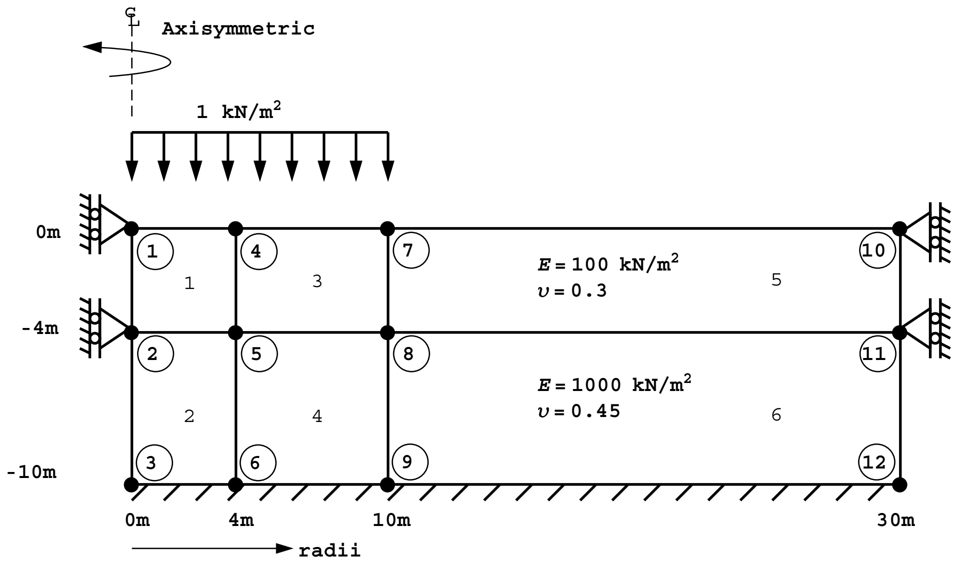


Figure 5.17 Mesh and data for fifth Program 5.1 example (*Continued on page 183*)

```
type_2d
'axisymmetric'

element          nod  dir
'quadrilateral'  4    'z'

nxe  nye  nip  np_types
3    2    9    2

prop(e,v)
 100.0  0.3
1000.0  0.45

etype
1 2 1 2 1 2

x_coords, y_coords
0.0   4.0  10.0  30.0
0.0  -4.0 -10.0

nr,(k,nf(:,k),i=1,nr)
8
1 0 1  2 0 1  3 0 0  6 0 0  9 0 0  10 0 1  11 0 1  12 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
3
1 0.0  -2.6667   4 0.0 -23.3333   7 0.0 -24.0

fixed_freedoms
0
```

Figure 5.17 (*Continued from page 182*)

```
There are   12 equations and the skyline storage is   58

 Node   r-disp      z-disp
   1  0.0000E+00 -0.3176E-01
   2  0.0000E+00 -0.3231E-02
   3  0.0000E+00  0.0000E+00
   4  0.1395E-02 -0.3990E-01
   5  0.1165E-02 -0.2498E-02
   6  0.0000E+00  0.0000E+00
   7  0.1704E-02 -0.6046E-02
   8  0.1330E-02 -0.4421E-03
   9  0.0000E+00  0.0000E+00
  10  0.0000E+00  0.2588E-02
  11  0.0000E+00  0.3091E-03
  12  0.0000E+00  0.0000E+00

The integration point (nip= 1) stresses are:
Element r-coord     z-coord     sig_r       sig_z       tau_rz      sig_t
   1  0.2000E+01 -0.2000E+01 -0.4140E+00 -0.1073E+01 -0.3452E-01 -0.4140E+00
   2  0.2000E+01 -0.7000E+01 -0.4776E+00 -0.9072E+00  0.6508E-01 -0.4776E+00
   3  0.7000E+01 -0.2000E+01 -0.2933E+00 -0.7099E+00  0.1180E+00 -0.2810E+00
   4  0.7000E+01 -0.7000E+01 -0.4316E+00 -0.6101E+00  0.1308E+00 -0.3796E+00
   5  0.2000E+02 -0.2000E+01 -0.3200E-01 -0.5814E-01  0.1082E-01 -0.2325E-01
   6  0.2000E+02 -0.7000E+01 -0.1090E+00 -0.9367E-01  0.4470E-01 -0.7455E-01
```

Figure 5.18 Results from fifth Program 5.1 example

Program 5.2 Non-axisymmetric analysis of an axisymmetric elastic solid using 8-node rectangular quadrilaterals. Mesh numbered in r- or z-direction.

```
PROGRAM p52
!-------------------------------------------------------------------------
! Program 5.2 Non-axisymmetric analysis of an axisymmetric elastic solid
!             using 8-node rectangular quadrilaterals. Mesh numbered in
!             r- or z- direction.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,iflag,k,loaded_nodes,lth,ndim=2,ndof=24,nels,neq,nip=4,   &
   nod=8,nodof=3,nn,nprops=2,np_types,nr,nre,nst=6,nze
 REAL(iwp)::ca,chi,det,one=1.0_iwp,pi,radius,sa,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   num(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),dee(:,:),der(:,:),deriv(:,:), &
 eld(:),fun(:),gc(:),g_coord(:,:),jac(:,:),km(:,:),kv(:),loads(:),        &
 points(:,:),prop(:,:),r_coords(:),sigma(:),weights(:),z_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nre,nze,lth,iflag,chi,np_types
 CALL mesh_size(element,nod,nels,nn,nre,nze)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),         &
   dee(nst,nst),coord(nod,ndim),fun(nod),jac(ndim,ndim),eld(ndof),        &
   weights(nip),der(ndim,nod),deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),&
   sigma(nst),num(nod),g_num(nod,nels),g_g(ndof,nels),gc(ndim),           &
   r_coords(nre+1),z_coords(nze+1),prop(nprops,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
 READ(10,*)r_coords,z_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq))
 pi=ACOS(-one); chi=chi*pi/180.0_iwp; ca=COS(chi); sa=SIN(chi); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,r_coords,z_coords,coord,num,'r')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1
 CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------element stiffness integration and assembly--------
 CALL sample(element,points,weights); kv=zero
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der)
     CALL bmat_nonaxi(bee,radius,coord,deriv,fun,iflag,lth)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)*radius
   END DO gauss_pts_1
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
!-----------------------equation solution---------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 WRITE(11,'(/A)')"  Node   r-disp      z-disp      t-disp"
 DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----------------------recover stresses at nip integrating points--------
 nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
 CALL sample(element,points,weights)
 WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
 WRITE(11,'(A,A)')" Element      r-coord     z-coord",                     &
   "     sig_r       sig_z       sig_t"
 WRITE(11,'(A,A)')"                                  ",                     &
   "     tau_rz      tau_zt      tau_tr"
 elements_3: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
   int_pts_2: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
     deriv=MATMUL(jac,der)
     CALL bmat_nonaxi(bee,radius,coord,deriv,fun,iflag,lth)
     bee(1:4,:)=bee(1:4,:)*ca; bee(5:6,:)=bee(5:6,:)*sa
     sigma=MATMUL(dee,MATMUL(bee,eld))
     WRITE(11,'(I5,5X,5E12.4)')iel,gc,sigma(:3)
     WRITE(11,'(34X,3E12.4)')sigma(4:6)
   END DO int_pts_2
 END DO elements_3
 CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p52
```

New scalar integers:

`iflag`	1 for “symmetry”, -1 for “antisymmetry”
`lth`	harmonic on which loads are to be applied
`nre`	number of elements in r-direction
`nze`	number of elements in z-direction

New scalar reals:

`ca`	set to `cos(chi)`
`chi`	angle for stress output
`pi`	set to π
`radius`	r-coordinate of Gauss point
`sa`	set to `sin(chi)`

New dynamic real arrays:

`r_coords`	r-coordinates of mesh layout
`z_coords`	z-coordinates of mesh layout

This program allows analysis of axisymmetric solids subjected to non-axisymmetric loads. Variations in displacements, and hence strains and stresses, tangentially are described by Fourier series (Wilson, 1965; Zienkiewicz and Taylor, 1989). Although the analysis is genuinely three dimensional, with 3 degrees of freedom at each node, it is only necessary to discretise the problem in a radial plane. The integrals in radial planes are performed using Gaussian quadrature in the usual way. Orthogonality relationships between typical terms in the tangential direction enable the integrals in the third direction to be stated explicitly. The problem therefore takes on the appearance of a two-dimensional analysis with the obvious benefits in terms of storage requirements. The disadvantages of the method over conventional three-dimensional finite element analysis are that, (1) the method is restricted to axisymmetric solids, and (2) for complicated loading distributions, several loading harmonic terms may be required, and a global stiffness matrix must be stored for each. Several harmonic terms may be required for elastic–plastic analysis (Griffiths, 1986), but for most elastic analyses such as the one described here, one harmonic will often be sufficient.

It is important to realise that the basic stiffness relationships relate *amplitudes* of load to *amplitudes* of displacement. Once the amplitudes of a displacement are known, the actual displacement at a particular circumferential location is easily found.

For simplicity, consider only the components of nodal load which are symmetric about the $\theta = 0$ axis of the axisymmetric body. In this case a general loading distribution may be given by

$$\begin{aligned} R &= \tfrac{1}{2}\overline{R}^{o} + \overline{R}^{1}\cos\theta + \overline{R}^{2}\cos 2\theta + \cdots \\ Z &= \tfrac{1}{2}\overline{Z}^{o} + \overline{Z}^{1}\cos\theta + \overline{Z}^{2}\cos 2\theta + \cdots \\ T &= \overline{T}^{1}\sin\theta + \overline{T}^{2}\sin 2\theta + \cdots \end{aligned} \tag{5.1}$$

where R, Z and T represent the load per radian in the radial, depth, and tangential directions. The bar terms with their superscripts, represent amplitudes of these quantities on the various harmonics.

For antisymmetric loading, symmetrical about the $\theta = \pi/2$ axis, these expressions become

$$\begin{aligned} R &= \overline{R}^{1}\sin\theta + \overline{R}^{2}\sin 2\theta + \cdots \\ Z &= \overline{Z}^{1}\sin\theta + \overline{Z}^{2}\sin 2\theta + \cdots \\ T &= \tfrac{1}{2}\overline{T}^{o} + \overline{T}^{1}\cos\theta + \overline{T}^{2}\cos 2\theta + \cdots \end{aligned} \tag{5.2}$$

Corresponding to these quantities are amplitudes of displacement in the radial, depth, and tangential directions. Since there are now three displacements per node, there are six strains at any point taken in the order,

$$\texttt{eps} = \begin{bmatrix} \epsilon_r & \epsilon_z & \epsilon_\theta & \gamma_{rz} & \gamma_{z\theta} & \gamma_{\theta r} \end{bmatrix}^{\mathrm{T}}$$

and six corresponding stresses, thus the 6×6 stress–strain matrix `dee` (2.84) is formed by the subroutine `deemat` as usual. Using the notation of equation (2.79), the **[A]** matrix now becomes,

$$[\mathbf{A}] == \begin{bmatrix} \frac{\partial}{\partial r} & 0 & 0 \\ 0 & \frac{\partial}{\partial z} & 0 \\ \frac{1}{r} & 0 & \frac{1}{r}\frac{\partial}{\partial \theta} \\ \frac{\partial}{\partial z} & \frac{\partial}{\partial r} & 0 \\ 0 & \frac{1}{r}\frac{\partial}{\partial \theta} & \frac{\partial}{\partial z} \\ \frac{1}{r}\frac{\partial}{\partial \theta} & 0 & \frac{\partial}{\partial r} - \frac{1}{r} \end{bmatrix} \tag{5.3}$$

For each harmonic i, the strain-displacement relationship provided by the library subroutine `bmat_nonaxi` is of the form,

$$\mathbf{B}^i = \left[\mathbf{B}_1^i \, \mathbf{B}_2^i \, \mathbf{B}_3^i \, \mathbf{B}_4^i \cdots \mathbf{B}_j^i \cdots \mathbf{B}_{\text{nod}}^i \right]$$

where `nod` is the number of nodes in an element.

A typical submatrix from the above expression for symmetric loading is given by

$$[\mathbf{B}]_j^i = \begin{bmatrix} \frac{\partial N_j}{\partial r}\cos i\theta & 0 & 0 \\ 0 & \frac{\partial N_j}{\partial z}\cos i\theta & 0 \\ \frac{N_j}{r}\cos i\theta & 0 & \frac{iN_j}{r}\cos i\theta \\ \frac{\partial N_j}{\partial z}\cos i\theta & \frac{\partial N_j}{\partial r}\cos i\theta & 0 \\ 0 & -\frac{iN_j}{r}\sin i\theta & \frac{\partial N_j}{\partial z}\sin i\theta \\ -\frac{iN_j}{r}\sin i\theta & 0 & \left(\frac{\partial N_j}{\partial r} - \frac{N_j}{r}\right)\sin i\theta \end{bmatrix} \tag{5.4}$$

The equivalent expression for antisymmetry is similar to equation (5.6) but with the sine and cosine terms interchanged and the signs of elements (3,3), (5,2) and (6,1) reversed. Additional `INTEGER` variables required by this subroutine are `iflag` and `lth`. The variable `iflag` is set to 1 or -1 for symmetry or antisymmetry respectively, and the variable `lth` gives the harmonic on which loads are to be applied. An additional variable input to this program is the angle `chi` (in degrees in the range 0° to 360°). This is the angle at which stresses are evaluated and printed. Naturally, stresses could be printed at other locations if required. It should be noted that if `lth=0` and `iflag=1`, the analysis reduces to ordinary axisymmetry as demonstrated by the fifth example with Program 5.1.

The program uses 8-node quadrilateral elements and can be considered a variant of Program 5.1 with nodes and elements numbered in the `dir='r'` direction. Each element has 24 degrees of freedom, as shown in Figure 5.19 and reduced integration (`nip=4`) is assumed.

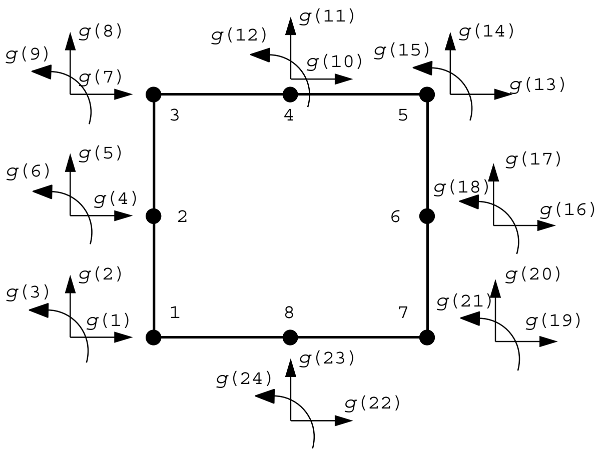


Figure 5.19 Local node and freedom numbering for the 8-node quadrilateral (three freedoms per node)

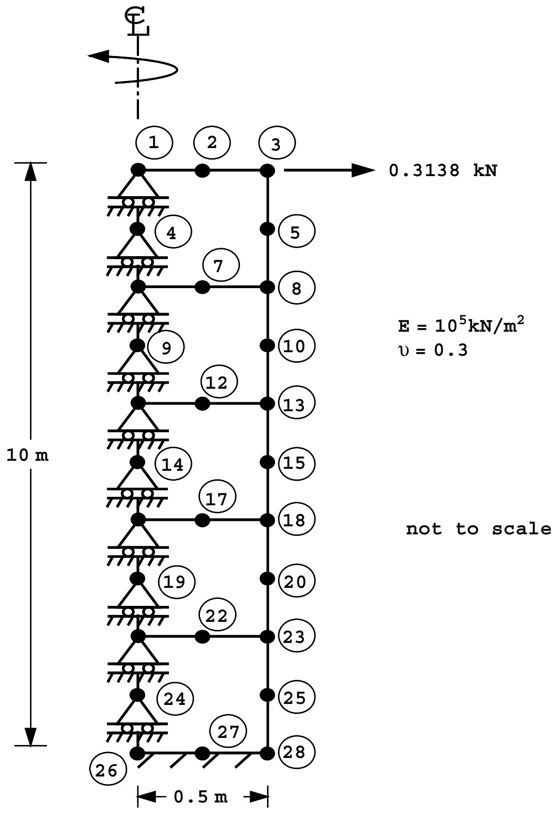


Figure 5.20 Mesh and data for Program 5.2 example (*Continued on page 189*)

```
nre  nze
1     5

lth iflag  chi  np_types
1     1    0.0     1

prop(e,v)
1.0e5  0.3

etype(not needed)

r_coords, z_coords
 0.0  0.5
10.0  8.0  6.0  4.0  2.0  0.0

nr,(k,nf(:,k),i=1,nr)
13
 1 1 0 1   4 1 0 1   6 1 0 1   9 1 0 1  11 1 0 1
14 1 0 1  16 1 0 1  19 1 0 1  21 1 0 1  24 1 0 1
26 0 0 0  27 0 0 0  28 0 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
1
3  0.3183  0.0  0.0
```

Figure 5.20 (*Continued from page 188*)

The example shown in Figure 5.20 represents a homogeneous cylindrical cantilever subjected to a transverse force of 1 kN at its tip. The nature of harmonic loading is such that a radial load amplitude of 1 unit on the first harmonic (`lth=1`) in symmetry (`iflag=1`) results in a net thrust in the $\theta = 0^\circ$ direction of π. Thus the load amplitude applied at the first freedom of node 3 equals $1/\pi$. Along the neutral axis, the nodal freedom data takes account of the fact that there can be no vertical movement along the centreline; hence these freedoms are restrained. The computed displacements in Figure 5.21 give the end

```
There are   65 equations and the skyline storage is  871

 Node   r-disp      z-disp      t-disp
   1  0.6755E-01  0.0000E+00 -0.6755E-01
   2  0.6755E-01 -0.2528E-02 -0.6755E-01
   3  0.6755E-01 -0.5063E-02 -0.6754E-01
   4  0.5743E-01  0.0000E+00 -0.5743E-01
   5  0.5744E-01 -0.5006E-02 -0.5743E-01
   6  0.4753E-01  0.0000E+00 -0.4752E-01
   7  0.4753E-01 -0.2426E-02 -0.4752E-01
   8  0.4753E-01 -0.4858E-02 -0.4750E-01
   9  0.3801E-01  0.0000E+00 -0.3801E-01
  10  0.3804E-01 -0.4598E-02 -0.3799E-01
.
.
.
  24  0.9378E-03  0.0000E+00 -0.9620E-03
  25  0.1052E-02 -0.9219E-03 -0.8806E-03
  26  0.0000E+00  0.0000E+00  0.0000E+00
  27  0.0000E+00  0.0000E+00  0.0000E+00
  28  0.0000E+00  0.0000E+00  0.0000E+00
```

Figure 5.21 Results from Program 5.2 example (*Continued on page 190*)

```
 The integration point (nip= 1) stresses are:
 Element      r-coord     z-coord     sig_r       sig_z       sig_t
                                      tau_rz      tau_zt      tau_tr
    1       0.2500E+00  0.9000E+01  0.6441E+00 -0.5036E+01 -0.4726E+00
                                    0.8638E-01  0.0000E+00  0.0000E+00
    2       0.2500E+00  0.7000E+01  0.2661E+01 -0.1413E+02  0.1144E+01
                                    0.6800E-01  0.0000E+00  0.0000E+00
    3       0.2500E+00  0.5000E+01  0.5441E+01 -0.2274E+02  0.3341E+01
                                    0.1484E+00  0.0000E+00  0.0000E+00
    4       0.2500E+00  0.3000E+01  0.8040E+01 -0.3164E+02  0.5250E+01
                                    0.3627E+00  0.0000E+00  0.0000E+00
    5       0.2500E+00  0.1000E+01  0.1700E+02 -0.3521E+02  0.1580E+02
                                    0.9873E+00  0.0000E+00  0.0000E+00
```

Figure 5.21 (*Continued from page 189*)

deflection of the cantilever to be 6.755×10^{-2} m, compared with the slender beam value of 6.791×10^{-2} m. If the same load amplitude was applied to the second freedom of node 3, it would correspond to a net moment of 0.5 kNm. The computed displacement in this case would be -5.063×10^{-3} m, compared with the slender beam value of -5.093×10^{-3} m. It should be noted that the current version of Program 5.2 is restricted to load control only.

Program 5.3 Three-dimensional analysis of an elastic solid using 8-, 14-, or 20-node brick hexahedra. Mesh numbered in x-z planes then in the y-direction.

```
PROGRAM p53
!-------------------------------------------------------------------------
! Program 5.3 Three-dimensional analysis of an elastic solid using
!             8-, 14- or 20-node brick hexahedra. Mesh numbered in x-z
!             planes then in the y-direction.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim=3,ndof,nels,neq,nip,nn,&
   nprops=2,np_types,nod,nodof=3,nr,nst=6,nxe,nye,nze
 REAL(iwp)::det,penalty=1.0e20_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::element='hexahedron'
!----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),dee(:,:),der(:,:),deriv(:,:), &
   eld(:),fun(:),gc(:),g_coord(:,:),jac(:,:),km(:,:),kv(:),loads(:),      &
   points(:,:),prop(:,:),sigma(:),value(:),weights(:),x_coords(:),        &
   y_coords(:),z_coords(:)
!----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nod,nxe,nye,nze,nip,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye,nze); ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),coord(nod,ndim),     &
   jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),g(ndof),bee(nst,ndof),    &
   km(ndof,ndof),eld(ndof),sigma(nst),g_g(ndof,nels),g_coord(ndim,nn),    &
   g_num(nod,nels),weights(nip),num(nod),prop(nprops,np_types),           &
   x_coords(nxe+1),y_coords(nye+1),z_coords(nze+1),etype(nels),fun(nod),  &
   gc(ndim))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords,z_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(loads(0:neq),kdiag(neq)); kdiag=0
```

```
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL hexahedron_xz(iel,x_coords,y_coords,z_coords,coord,num)
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                  &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------element stiffness integration and assembly--------
 CALL sample(element,points,weights); kv=zero
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
   num=g_num(:,iel); g=g_g(:,iel); coord=TRANSPOSE(g_coord(:,num)); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i); jac=MATMUL(der,coord)
     det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
     CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1
   CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
 loads=zero; READ(10,*)loaded_nodes
 IF(loaded_nodes/=0)READ(10,*)(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                 &
     value(fixed_freedoms),no(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
!-----------------------equation solution----------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 WRITE(11,'(/A)')"  Node   x-disp      y-disp      z-disp"
 DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----------------------recover stresses at nip integrating points--------
 nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
 CALL sample(element,points,weights)
 WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
 WRITE(11,'(A,/,A)')"    Element     x-coord     y-coord     z-coord",    &
   "    sig_x       sig_y       sig_z       tau_xy      tau_yz      tau_zx"
 elements_3: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
   gauss_pts_2: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     sigma=MATMUL(dee,MATMUL(bee,eld)); WRITE(11,'(I8,4X,3E12.4)')iel,gc
     WRITE(11,'(6E12.4)')sigma
   END DO gauss_pts_2
 END DO elements_3
STOP
END PROGRAM p53
```

In cases where many Fourier harmonics are required to define a loading pattern, it becomes more efficient and certainly simpler, to solve the full three-dimensional problem.

Program 5.3 is the 3D counterpart of Program 5.1, and is capable of performing elastic analysis of 3D cuboid meshes consisting of hexahedral brick-shaped elements. The program can incorporate 8-, 14-, or 20-node hexahedra and includes a geometry subroutine called `hexahedron_xz` for generating meshes in which nodes and elements are counted in *xz*-planes moving in the *y*-direction as illustrated in Figure 5.22.

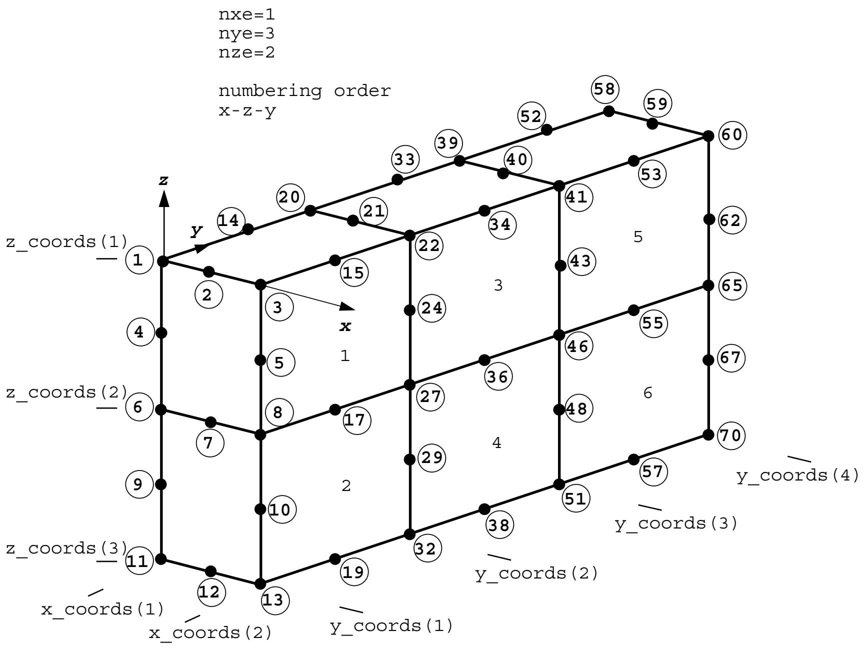


Figure 5.22 Global node and element numbering for mesh of 20-node hexahedra

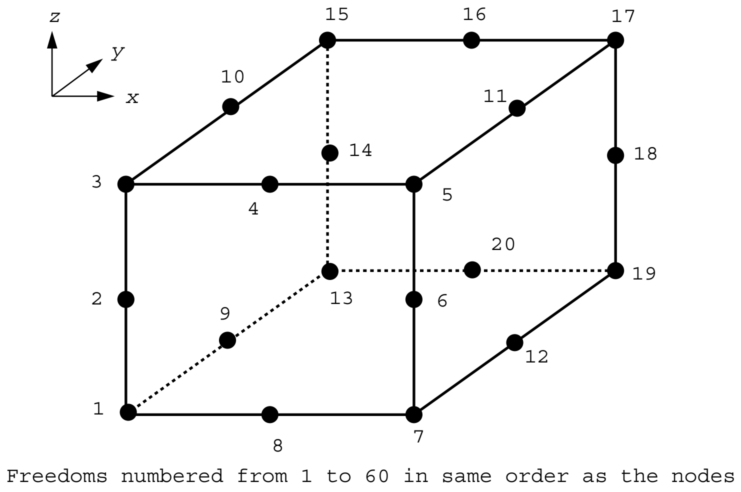
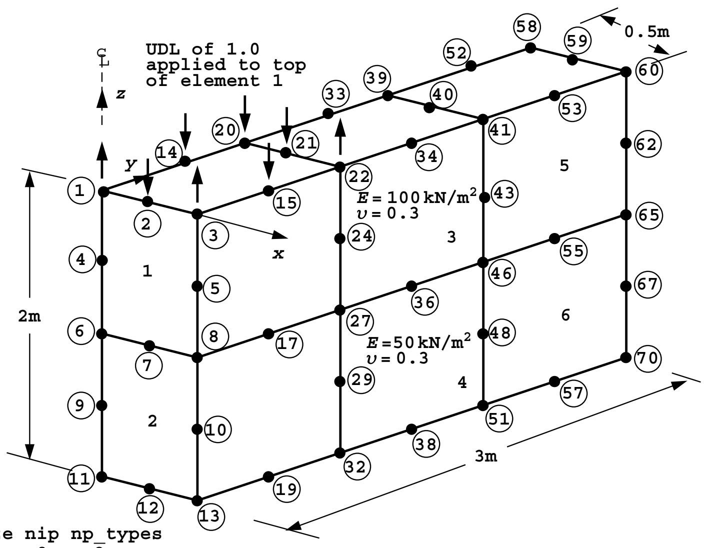


Figure 5.23 Local node and freedom numbering for the 20-node hexahedral element

A widely used three-dimensional element, the 20-node hexahedron, is the subject of the example that goes with this program. The element is the three-dimensional analogue of the 8-noded quadrilateral in plane problems with the element node numbering indicated in Figure 5.23.

The example and data of Figure 5.24 are for a simple boundary value problem where an elastic block carries a unit uniform load over part of its top surface. The block has



```
nod
20

nxe nye nze nip np_types
1    3   2   8     2

prop(e,v)
100.0  0.3
 50.0  0.3

etype
1 2 1 2 1 2

x_coords, y_coords, z_coords
 0.0  0.5
 0.0  1.0  2.0  3.0
 0.0 -1.0 -2.0

nr,(k,nf(:,k),i=1,nr)
46
 1 0 0 1   2 1 0 1   3 1 0 1   4 0 0 1   5 1 0 1   6 0 0 1
 7 1 0 1   8 1 0 1   9 0 0 1  10 1 0 1  11 0 0 0  12 0 0 0
13 0 0 0  14 0 1 1  16 0 1 1  18 0 0 0  19 0 0 0  20 0 1 1
23 0 1 1  25 0 1 1  28 0 1 1  30 0 0 0  31 0 0 0  32 0 0 0
33 0 1 1  35 0 1 1  37 0 0 0  38 0 0 0  39 0 1 1  42 0 1 1
44 0 1 1  47 0 1 1  49 0 0 0  50 0 0 0  51 0 0 0  52 0 1 1
54 0 1 1  56 0 0 0  57 0 0 0  58 0 1 1  61 0 1 1  63 0 1 1
66 0 1 1  68 0 0 0  69 0 0 0  70 0 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
8
 1  0.0  0.0  0.0417    2  0.0  0.0 -0.1667    3  0.0  0.0  0.0417
14  0.0  0.0 -0.1667   15  0.0  0.0 -0.1667   20  0.0  0.0  0.0417
21  0.0  0.0 -0.1667   22  0.0  0.0  0.0417

fixed_freedoms
0
```

Figure 5.24 Mesh and data for Program 5.3 example

layered properties, with the upper and lower halves assigned Young's moduli values of 100 and 50 respectively. Poisson's ratio is fixed at 0.3.

An exact integration scheme for 20-node cuboid elements would need `nip=27`, however "reduced integration" is recommended for this element, thus `nip=8` in the present analysis. The reader can also experiment with Irons's (1971) approximate integration rules, for example `nip` can be set to 6, 14, and 15. The nodal forces to simulate a unit uniform stress field for this element are certainly not intuitive, and involve corner loads which act in the opposite direction to the mid-side loads (Appendix A).

The computed displacements and centroid stresses are given in Figure 5.25. For example the z-deflection at the origin of the coordinate system (node 1) is computed as -0.2246×10^{-1}.

Program 5.3 uses conventional storage and solution strategies, however storage requirements for 3D analysis rapidly become substantial. It can be noted that even in a simple example such as this, the skyline stiffness vector `kv` still requires 4388 locations. For this

```
 There are  124 equations and the skyline storage is 4388

  Node   x-disp      y-disp      z-disp
    1  0.0000E+00  0.0000E+00 -0.2246E-01
    2  0.1584E-02  0.0000E+00 -0.2255E-01
    3  0.3220E-02  0.0000E+00 -0.2333E-01
    4  0.0000E+00  0.0000E+00 -0.1849E-01
    5  0.1544E-02  0.0000E+00 -0.1884E-01
    6  0.0000E+00  0.0000E+00 -0.1443E-01
    7  0.7581E-03  0.0000E+00 -0.1435E-01
    8  0.1511E-02  0.0000E+00 -0.1411E-01
    9  0.0000E+00  0.0000E+00 -0.6164E-02
   10  0.2792E-02  0.0000E+00 -0.6430E-02
.
.
.
   66  0.0000E+00  0.1572E-02 -0.1028E-03
   67 -0.7448E-04  0.1716E-02 -0.5846E-04
   68  0.0000E+00  0.0000E+00  0.0000E+00
   69  0.0000E+00  0.0000E+00  0.0000E+00
   70  0.0000E+00  0.0000E+00  0.0000E+00

 The integration point (nip= 1) stresses are:
    Element     x-coord     y-coord     z-coord
    sig_x       sig_y       sig_z       tau_xy      tau_yz      tau_zx
        1       0.2500E+00  0.5000E+00 -0.5000E+00
 -0.2672E-01 -0.1647E+00 -0.9088E+00  0.6145E-02  0.9597E-01  0.4352E-02
        2       0.2500E+00  0.5000E+00 -0.1500E+01
  0.3985E-01 -0.5316E-01 -0.6298E+00 -0.2140E-02  0.7614E-01  0.4169E-02
        3       0.2500E+00  0.1500E+01 -0.5000E+00
 -0.2482E-01 -0.1260E+00 -0.1052E+00  0.4840E-02  0.9399E-01 -0.2814E-02
        4       0.2500E+00  0.1500E+01 -0.1500E+01
  0.2477E-01 -0.8240E-01 -0.2822E+00 -0.3179E-02  0.1214E+00  0.2939E-02
        5       0.2500E+00  0.2500E+01 -0.5000E+00
  0.3767E-02  0.8619E-02 -0.1469E-01 -0.9006E-03 -0.8733E-02  0.8652E-03
        6       0.2500E+00  0.2500E+01 -0.1500E+01
  0.7407E-02 -0.4390E-01 -0.2831E-01 -0.5639E-03  0.6083E-01  0.5078E-04
```

Figure 5.25 Results from Program 5.3 example

reason, later programs in this Chapter, Programs 5.5 and 5.6, introduce solution methods in which assembly of the global stiffness matrix is avoided entirely.

Program 5.4 General two- (plane strain) or three-dimensional analysis of elastic solids.

```
PROGRAM p54
!-------------------------------------------------------------------------
! Program 5.4 General two- (plane strain) or three-dimensional analysis
!             of elastic solids (optional gravity loading).
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim,ndof,nels,neq,nip,nn,  &
   nod,nodof,nprops=3,np_types,nr,nst
 REAL(iwp)::det,penalty=1.0e20_iwp,zero=0.0_iwp
 CHARACTER(len=15)::element
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),dee(:,:),der(:,:),deriv(:,:), &
   eld(:),fun(:),gc(:),gravlo(:),g_coord(:,:),jac(:,:),km(:,:),kv(:),     &
   loads(:),points(:,:),prop(:,:),sigma(:),value(:),weights(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)element,nod,nels,nn,nip,nodof,nst,ndim,np_types; ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),g_coord(ndim,nn),    &
   coord(nod,ndim),jac(ndim,ndim),weights(nip),num(nod),g_num(nod,nels),  &
   der(ndim,nod),deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),eld(ndof),   &
   sigma(nst),g(ndof),g_g(ndof,nels),gc(ndim),fun(nod),etype(nels),       &
   prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)g_coord; READ(10,*)g_num
 IF(ndim==2)CALL mesh(g_coord,g_num,12)
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),gravlo(0:neq)); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------element stiffness integration and assembly--------
 CALL sample(element,points,weights); kv=zero; gravlo=zero
 elements_2: DO  iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; eld=zero
   int_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(transpose(bee),dee),bee)*det*weights(i)
     eld(nodof:ndof:nodof)=eld(nodof:ndof:nodof)+fun(:)*det*weights(i)
   END DO int_pts_1
   CALL fsparv(kv,km,g,kdiag); gravlo(g)=gravlo(g)-eld*prop(3,etype(iel))
 END DO elements_2
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 loads=loads+gravlo; READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                  &
     value(fixed_freedoms),no(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO  i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
 END IF
!-----------------------equation solution---------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
 IF(ndim==3)THEN; WRITE(11,'(/A)')"  Node   x-disp      y-disp      z-disp"
 ELSE; WRITE(11,'(/A)')"  Node   x-disp      y-disp"
 END IF
 DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----------------------recover stresses at element Gauss-points----------
!nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
!CALL sample(element,points,weights)
 WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
 IF(ndim==3)THEN
   WRITE(11,'(A,/,A)')"    Element     x-coord     y-coord     z-coord",  &
   "    sig_x       sig_y       sig_z       tau_xy      tau_yz      tau_zx"
 ELSE; WRITE(11,'(A,A)')  "    Element x-coord     y-coord",               &
   "          sig_x       sig_y       tau_xy"
 END IF
 elements_3: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
   int_pts_2: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     sigma=MATMUL(dee,MATMUL(bee,eld))
     IF(ndim==3)THEN; WRITE(11,'(I8,4X,3E12.4)')iel,gc
       WRITE(11,'(6E12.4)')sigma
     ELSE; WRITE(11,'(I8,2E12.4,5X,3E12.4)')iel,gc,sigma
     END IF
   END DO int_pts_2
 END DO elements_3
 IF(ndim==2)THEN; CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
   CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
 END IF
STOP
END PROGRAM p54
```

New dynamic real arrays:

`gravlo`	loads generated by gravity

Perusal of Programs 5.1 and 5.3 in this chapter will show that they are essentially identical. The shape function and derivative subroutines `shape_fun` and `shape_der`, and the [**B**] and [**D**] subroutines `beemat` and `deemat` can all generate the appropriate terms once the 2 or 3D element type has been identified through the data. Program 5.4 utilises this identity of programs to create a single general program. Of course such a program will expect to read the nodal geometry `g_coord` and connectivity `g_num` details from a file which would usually be provided by a mesh-generation pre-processor. The element types available in the library are, for plane strain or axisymmetry:

3, 6, 10, and 15-node triangles
4, 8, and 9-node quadrilaterals

and for 3D:

4-node tetrahedra
8, 14, and 20-node hexahedra.

The local numbering of all these elements which is needed for input to Program 5.4 is given in Appendix B. In addition to the usual two elastic parameters of Young's modulus and Poisson's ratio, this program also allows the option of gravity load generation through the unit weight, which must be read in as a third property (`nprops=3`) for each property group. If gravity loading is not required, the unit weight should be read as zero.

The global gravity loading vector (called `gravlo` in the program) for a material with unit weight γ is accumulated from each element by integration of the shape functions $[\mathbf{N}]$ as follows,

$$\texttt{gravlo} = \sum_{elements}^{all} \gamma \iint [\mathbf{N}]^{\mathrm{T}}\, \mathrm{d}x\, \mathrm{d}y \tag{5.5}$$

and these calculations are performed in the same part of the program that forms the global stiffness matrix. It may be noted that only those freedoms corresponding to vertical movement are incorporated in the integrals. At the element level, the one-dimensional array `eld` is used to gather the contributions from each Gauss point. The global gravity loads vector `gravlo` accumulates `eld` from each element after multiplication by the unit weight γ held in the `prop` array.

The first example uses a 9-node "Lagrangian" element with numbering shown in Figure 5.26. The example problem in plane strain shown in Figure 5.27 is deliberately chosen to allow a comparison with a similar problem previously analysed by Program 5.1 using 8-node elements (Figures 5.15 and 5.16). No gravitational loading has been included in this example, and "exact" integration has been used by setting `nip` equal to 9. This version of Program 5.4 elects to print stresses at all the integrating points used in the stiffness integration, thus the output file prints stresses at 9 locations per element. The results given in Figure 5.28 indicate a centreline displacement of -0.5299×10^{-5} m, and a centroid stress σ_y in the first element (also located at the central node) of $-0.8766\,\mathrm{kN/m^2}$. This could be compared with the centreline displacement of -0.5311×10^{-5} and centroid stress of -0.9003 from Figure 5.16 using the 8-node element. Users can experiment with the influence of gravity. In this example, if the third property representing the unit weight is set to 1.0, and `loaded_nodes` is set to zero (gravity loading only), the vertical stress σ_y is computed to be identical to the depth of each integrating point.

The second example illustrates the use of the simplest 3D element, namely the 4-node tetrahedron with numbering given in Figure 5.29. This "constant-strain" element is

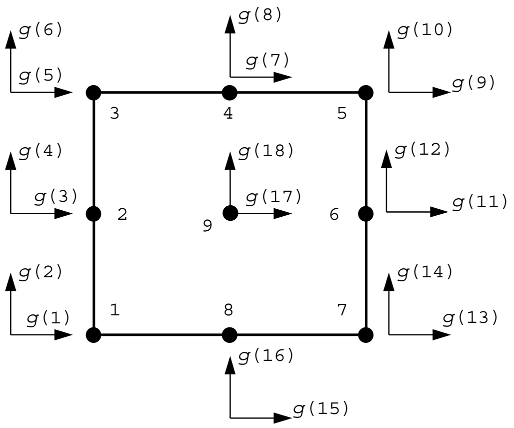


Figure 5.26 Local node and freedom numbering for the 9-node quadrilateral

analogous to the 3-noded triangle for plane problems described in Program 5.1 and, like the triangle, is not recommended for practical calculations unless adaptive mesh refinement has been implemented. Like the 3-node triangle, this element is exactly integrated using `nip=1`. The example in Figure 5.30 represents a homogeneous cube made up of six tetrahedra. One corner of the cube is fixed and the three adjacent faces are restrained to move only in their own planes. The four nodal forces applied are equivalent to a uniform vertical

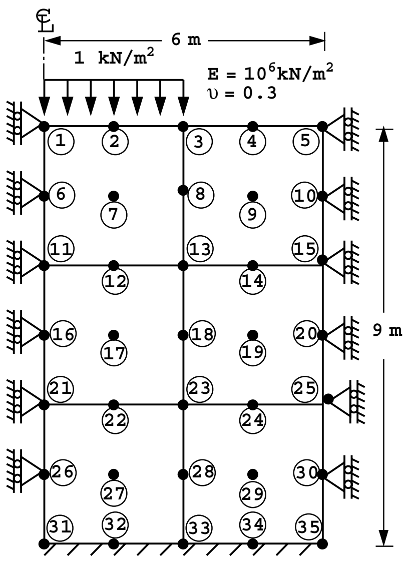


Figure 5.27 Mesh and data for first Program 5.4 example (*Continued on page 199*)

```
element        nod
'quadrilateral' 9

nels nn nip nodof nst ndim np_types
6    35  9    2    3   2     1

prop(e,v,γ)
1.0e6  0.3  0.0

etype(not needed)

g_coord
0.0  0.0   1.5  0.0   3.0  0.0   4.5  0.0   6.0  0.0   0.0 -1.5
1.5 -1.5   3.0 -1.5   4.5 -1.5   6.0 -1.5   0.0 -3.0   1.5 -3.0
3.0 -3.0   4.5 -3.0   6.0 -3.0   0.0 -4.5   1.5 -4.5   3.0 -4.5
4.5 -4.5   6.0 -4.5   0.0 -6.0   1.5 -6.0   3.0 -6.0   4.5 -6.0
6.0 -6.0   0.0 -7.5   1.5 -7.5   3.0 -7.5   4.5 -7.5   6.0 -7.5
0.0 -9.0   1.5 -9.0   3.0 -9.0   4.5 -9.0   6.0 -9.0

g_num
11   6   1   2   3   8  13  12   7
21  16  11  12  13  18  23  22  17
31  26  21  22  23  28  33  32  27
13   8   3   4   5  10  15  14   9
23  18  13  14  15  20  25  24  19
33  28  23  24  25  30  35  34  29

nr,(k,nf(:,k),i=1,nr)
17
 1 0 1    5 0 1    6 0 1   10 0 1   11 0 1   15 0 1
16 0 1   20 0 1   21 0 1   25 0 1   26 0 1   30 0 1
31 0 0   32 0 0   33 0 0   34 0 0   35 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
3
1 0.0 -0.5   2 0.0 -2.0   3 0.0 -0.5

fixed_freedoms
0
```

Figure 5.27 (*Continued from page 198*)

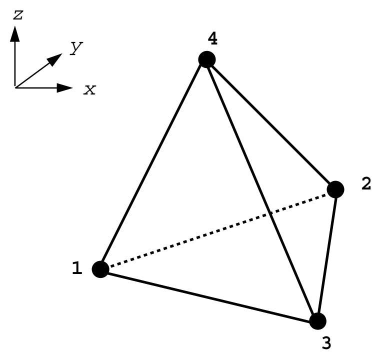
compressive stress of 1 kN/m^2 (Appendix A). The computed results given in Figure 5.31 show that the cube compresses uniformly and that the vertical stress σ_z, at the centroid is equal to unity, and in equilibrium with the applied loads.

The simplest member of the hexahedral or "brick" element family has 8 nodes, situated at the corners, however the element is quite "stiff" in certain deformation modes and a more commonly available element in commercial programs is the 20-node brick. Both of these elements are available in Programs 5.3 and 5.4, as is an intermediate 14-node element proposed by Smith and Kidger (1992). This intermediate element has 8 corner nodes, supplemented by 6 mid-face nodes with a numbering system given in Figure 5.32. There are several versions of this element, and one of them ("Type 6") is illustrated in the next example (See Section 3.7.9).

The analysis shown in Figure 5.33 is of a "patch" mesh suggested by Peano (1987) for testing the admissibility of solid elements. The outer cube has smooth, rigid boundary

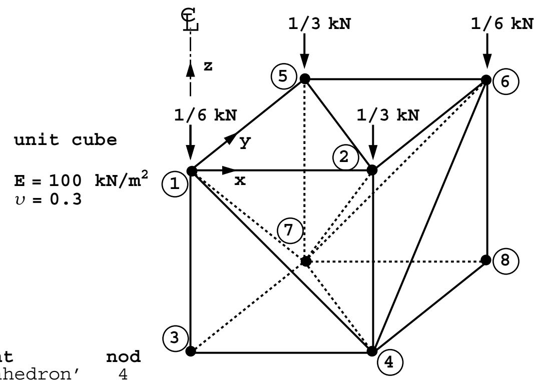
```
There are   48 equations and the skyline storage is  610

 Node   x-disp      y-disp
   1  0.0000E+00 -0.5299E-05
   2 -0.4004E-06 -0.4988E-05
   3 -0.6190E-06 -0.3343E-05
   4 -0.4004E-06 -0.1697E-05
   5  0.0000E+00 -0.1387E-05
   6  0.0000E+00 -0.4307E-05
   7  0.1856E-06 -0.3911E-05
   8  0.3167E-06 -0.2786E-05
   9  0.1856E-06 -0.1661E-05
  10  0.0000E+00 -0.1264E-05
.
.
.
  31  0.0000E+00  0.0000E+00
  32  0.0000E+00  0.0000E+00
  33  0.0000E+00  0.0000E+00
  34  0.0000E+00  0.0000E+00
  35  0.0000E+00  0.0000E+00

The integration point (nip= 9) stresses are:
   Element x-coord     y-coord          sig_x       sig_y       tau_xy
       1  0.3381E+00 -0.3381E+00      -0.6323E+00 -0.1035E+01 -0.4108E-01
       1  0.1500E+01 -0.3381E+00      -0.5703E+00 -0.1047E+01  0.5463E-01
       1  0.2662E+01 -0.3381E+00      -0.3507E+00 -0.6910E+00  0.1917E+00
       1  0.3381E+00 -0.1500E+01      -0.2434E+00 -0.9108E+00  0.2695E-01
       1  0.1500E+01 -0.1500E+01      -0.2597E+00 -0.8766E+00  0.1085E+00
       1  0.2662E+01 -0.1500E+01      -0.1681E+00 -0.5906E+00  0.2173E+00
       1  0.3381E+00 -0.2662E+01      -0.1395E+00 -0.9083E+00  0.5062E-01
       1  0.1500E+01 -0.2662E+01      -0.1846E+00 -0.8070E+00  0.1512E+00
       1  0.2662E+01 -0.2662E+01      -0.1714E+00 -0.5698E+00  0.2648E+00
.
.
.
       6  0.4500E+01 -0.6338E+01      -0.2436E+00 -0.4162E+00  0.4191E-01
       6  0.5662E+01 -0.6338E+01      -0.2740E+00 -0.3928E+00  0.1298E-01
       6  0.3338E+01 -0.7500E+01      -0.2245E+00 -0.4855E+00  0.4446E-01
       6  0.4500E+01 -0.7500E+01      -0.2283E+00 -0.4382E+00  0.2936E-01
       6  0.5662E+01 -0.7500E+01      -0.2449E+00 -0.4208E+00  0.8179E-02
       6  0.3338E+01 -0.8662E+01      -0.2137E+00 -0.4886E+00  0.3231E-01
       6  0.4500E+01 -0.8662E+01      -0.2059E+00 -0.4572E+00  0.2255E-01
       6  0.5662E+01 -0.8662E+01      -0.2063E+00 -0.4448E+00  0.6023E-02
```

Figure 5.28 Results from first Program 5.4 example



Freedoms numbered from 1 to 12 in same order as the nodes

Figure 5.29 Local node and freedom numbering for the 4-node tetrahedral element

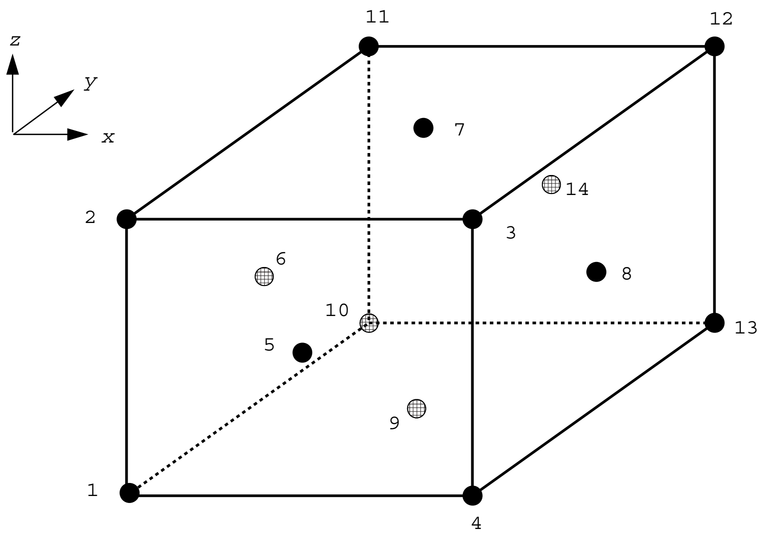


```
element        nod
'tetrahedron'   4

nels nn nip nodof nst ndim np_types
6     8  1    3    6   3     1

prop(e,v,γ)
100.0  0.3  0.0

etype(not needed)

g_coord
0.0  0.0  0.0   1.0  0.0  0.0   0.0  0.0 -1.0   1.0  0.0 -1.0
0.0  1.0  0.0   1.0  1.0  0.0   0.0  1.0 -1.0   1.0  1.0 -1.0

g_num
1 3 4 7   1 4 2 7   1 2 5 7
6 4 8 7   6 2 4 7   6 5 2 7

nr,(k,nf(:,k),i=1,nr)
7
1 0 0 1  2 1 0 1  3 0 0 0  4 1 0 0
5 0 1 1  7 0 1 0  8 1 1 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
4
1  0.0  0.0 -0.1667   2  0.0  0.0 -0.3333
5  0.0  0.0 -0.3333   6  0.0  0.0 -0.1667

fixed_freedoms
0
```

Figure 5.30 Mesh and data for second Program 5.4 example

```
There are   12 equations and the skyline storage is   69

 Node   x-disp      y-disp      z-disp
   1  0.0000E+00  0.0000E+00 -0.1000E-01
   2  0.3000E-02  0.0000E+00 -0.1000E-01
   3  0.0000E+00  0.0000E+00  0.0000E+00
   4  0.3000E-02  0.0000E+00  0.0000E+00
   5  0.0000E+00  0.3000E-02 -0.9999E-02
   6  0.3000E-02  0.3000E-02 -0.1000E-01
   7  0.0000E+00  0.3000E-02  0.0000E+00
   8  0.3000E-02  0.3000E-02  0.0000E+00

The integration point (nip= 1) stresses are:
   Element      x-coord     y-coord     z-coord
   sig_x        sig_y       sig_z       tau_xy      tau_yz      tau_zx
       1       0.2500E+00  0.2500E+00 -0.7500E+00
-0.1965E-04 -0.2100E-04 -0.1000E+01  0.0000E+00  0.0000E+00  0.0000E+00
       2       0.5000E+00  0.2500E+00 -0.5000E+00
 0.2302E-05  0.1389E-04 -0.1000E+01 -0.6475E-05  0.2974E-04  0.2326E-04
       3       0.2500E+00  0.5000E+00 -0.2500E+00
 0.1230E-04  0.2075E-05 -0.9999E+00  0.0000E+00  0.3641E-04  0.2974E-04
       4       0.7500E+00  0.7500E+00 -0.7500E+00
-0.1087E-04 -0.9023E-05 -0.1000E+01  0.1556E-05 -0.7467E-05 -0.9316E-05
       5       0.7500E+00  0.5000E+00 -0.5000E+00
 0.6102E-05 -0.1297E-05 -0.1000E+01 -0.8752E-05 -0.2541E-04 -0.3189E-04
       6       0.5000E+00  0.7500E+00 -0.2500E+00
 0.9809E-05  0.1536E-04 -0.9999E+00  0.2158E-05 -0.3632E-04 -0.4299E-04
```

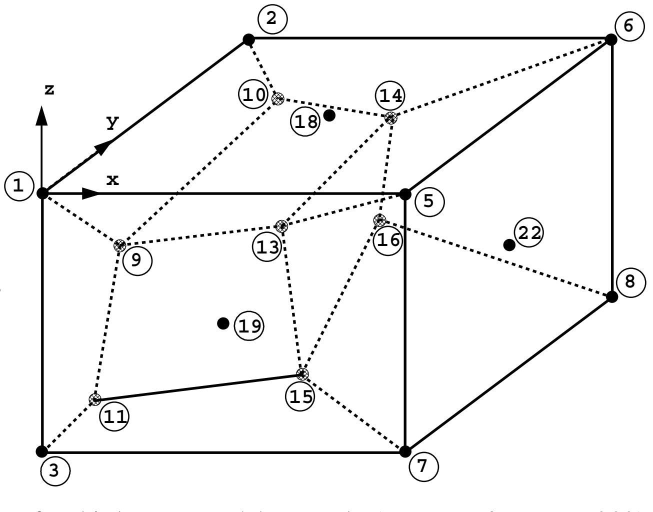
Figure 5.31 Results from second Program 5.4 example



Freedoms numbered from 1 to 42 in same order as the nodes

Figure 5.32 Local node and freedom numbering for the 14-node hexahedral element

conditions to the left ($x = 0$) and bottom ($z = -1$), and the 5 nodes on the front face of the exterior box are given a uniform y-displacement of 0.01. The results are shown in Figure 5.34 where it can be seen that a completely homogeneous strain field has resulted in a stress of $\sigma_y = -0.01\,\text{kN/m}^2$ at all integrating points within the mesh. Thus, the element passes the patch test. The current example used `nip=27`, but users could experiment with different orders of integration.



```
Figure shows visible
nodes on the surface,
plus corner nodes only
of inner element.
Outer "box" is
a unit cube

element        nod
'hexahedron'    14
```

Figure 5.33 Mesh and data for third Program 5.4 example (*Continued on page 203*)

```
nels nn nip nodof nst ndim np_types
7    40  27   3    6   3     1

prop(e,v,γ)
1.0  0.49  0.0

etype(not needed)

g_coord
0.0000  0.0000  0.0000     0.0000  1.0000  0.0000
0.0000  0.0000 -1.0000     0.0000  1.0000 -1.0000
.
. (g_coord data for nodes 5-36 omitted here)
.
0.1423  0.5412 -0.8680     0.5187  0.8800 -0.8425
0.8758  0.5072 -0.8518     0.4992  0.1685 -0.8773

g_num
11   9  13  15  25  23  24  28  27  12  10  14  16  26
 3   1   9  11  33  17  29  23  37   4   2  10  12  34
 9   1   5  13  32  29  18  31  24  10   2   6  14  30
15  13   5   7  36  28  31  22  39  16  14   6   8  35
 3  11  15   7  40  37  27  39  21   4  12  16   8  38
12  10  14  16  26  34  30  35  38   4   2   6   8  20
 3   1   5   7  19  33  32  36  40  11   9  13  15  25

nr,(k,nf(:,k),i=1,nr)
10
1 0 1 1   2 0 0 1   3 0 1 0   4 0 0 0   6 1 0 1
7 1 1 0   8 1 0 0  17 0 1 1  20 1 0 1  21 1 1 0

loaded_nodes
0

fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
5
1  2  0.01   3  2  0.01   5  2  0.01   7  2  0.01  19  2  0.01
```

Figure 5.33 *(Continued from page 202)*

```
    There are  105 equations and the skyline storage is 5487

     Node   x-disp      y-disp      z-disp
       1  0.0000E+00  0.1000E-01  0.4900E-02
       2  0.0000E+00  0.0000E+00  0.4900E-02
       3  0.0000E+00  0.1000E-01  0.0000E+00
       4  0.0000E+00  0.0000E+00  0.0000E+00
       5  0.4900E-02  0.1000E-01  0.4900E-02
       6  0.4900E-02  0.0000E+00  0.4900E-02
       7  0.4900E-02  0.1000E-01  0.0000E+00
       8  0.4900E-02  0.0000E+00  0.0000E+00
       9  0.8085E-03  0.7020E-02  0.3651E-02
      10  0.1333E-02  0.2300E-02  0.3675E-02
   .
   .
   .
      37  0.6973E-03  0.4588E-02  0.6468E-03
      38  0.2542E-02  0.1200E-02  0.7718E-03
      39  0.4291E-02  0.4928E-02  0.7262E-03
      40  0.2446E-02  0.8315E-02  0.6012E-03
```

Figure 5.34 Results from third Program 5.4 example *(Continued on page 204)*

```
 The integration point (nip=27) stresses are:
    Element     x-coord     y-coord     z-coord
    sig_x        sig_y       sig_z       tau_xy      tau_yz      tau_zx
       1      0.2589E+00  0.3618E+00 -0.3200E+00
 -0.1793E-07 -0.1000E-01 -0.1827E-07  0.1618E-09 -0.1391E-08 -0.2100E-09
       1      0.4879E+00  0.3762E+00 -0.3354E+00
 -0.5606E-08 -0.1000E-01 -0.6012E-08 -0.4430E-09 -0.1763E-08 -0.1400E-09
       1      0.7170E+00  0.3905E+00 -0.3508E+00
  0.1582E-08 -0.1000E-01 -0.7704E-10 -0.4557E-09 -0.8293E-09 -0.1176E-09
.
.
.
       7      0.4889E+00  0.2946E+00 -0.5157E+00
 -0.8196E-07 -0.1000E-01 -0.8207E-07 -0.2538E-09 -0.8485E-09 -0.2121E-09
       7      0.7010E+00  0.2977E+00 -0.5042E+00
 -0.3727E-08 -0.1000E-01 -0.3632E-08 -0.7091E-09  0.5663E-09 -0.1256E-09
       7      0.3198E+00  0.3080E+00 -0.7564E+00
  0.3311E-07 -0.1000E-01  0.3315E-07  0.1318E-08  0.7962E-09 -0.1938E-09
       7      0.4965E+00  0.2980E+00 -0.7221E+00
  0.1311E-07 -0.1000E-01  0.1332E-07  0.1924E-09  0.1619E-08  0.1072E-09
       7      0.6731E+00  0.2881E+00 -0.6878E+00
 -0.1021E-07 -0.1000E-01 -0.1057E-07  0.8205E-10  0.1063E-08  0.5315E-10
```

Figure 5.34 (*Continued from page 203*)

Program 5.5 Three-dimensional strain of an elastic solid using 8-, 14-, or 20-node brick hexahedra. Mesh numbered in *x*-*z* planes then in the *y*-direction. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver.

```
PROGRAM p55
!-------------------------------------------------------------------------
! Program 5.5 Three-dimensional strain of an elastic solid using
!             8-, 14- or 20-node brick hexahedra. Mesh numbered in x-z
!             planes then in the y-direction. No global stiffness matrix
!             assembly. Diagonally preconditioned conjugate gradient solver.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,k,loaded_nodes,ndim=3,   &
   ndof,nels,neq,nip,nn,nprops=2,np_types,nod,nodof=3,nr,nst=6,nxe,nye,nze
 REAL(iwp)::alpha,beta,cg_tol,det,one=1.0_iwp,penalty=1.0e20_iwp,up,      &
   zero=0.0_iwp
 CHARACTER(LEN=15)::element='hexahedron'; LOGICAL::cg_converged
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),nf(:,:),no(:),    &
   node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),d(:),dee(:,:),der(:,:),       &
   deriv(:,:),diag_precon(:),eld(:),fun(:),gc(:),g_coord(:,:),jac(:,:),   &
   km(:,:),loads(:),p(:),points(:,:),prop(:,:),sigma(:),store(:),         &
   storkm(:,:,:),u(:),value(:),weights(:),x(:),xnew(:),x_coords(:),       &
   y_coords(:),z_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nod,nxe,nye,nze,nip,cg_tol,cg_limit,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye,nze); ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),coord(nod,ndim),     &
   jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),fun(nod),gc(ndim),        &
   bee(nst,ndof),km(ndof,ndof),eld(ndof),sigma(nst),g_coord(ndim,nn),     &
   g_num(nod,nels),weights(nip),num(nod),g_g(ndof,nels),x_coords(nxe+1),  &
   g(ndof),y_coords(nye+1),z_coords(nze+1),etype(nels),                   &
   prop(nprops,np_types),storkm(ndof,ndof,nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords,z_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 WRITE(11,'(A,I5,A)')" There are",neq," equations"
 ALLOCATE(p(0:neq),loads(0:neq),x(0:neq),xnew(0:neq),u(0:neq),         &
   diag_precon(0:neq),d(0:neq))
 CALL sample(element,points,weights); diag_precon=zero
!----------element stiffness integration, storage and preconditioner------
 elements_1: DO iel=1,nels
   CALL hexahedron_xz(iel,x_coords,y_coords,z_coords,coord,num)
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
   num=g_num(:,iel); g=g_g(:,iel); coord=TRANSPOSE(g_coord(:,num)); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1
   storkm(:,:,iel)=km
   DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
 END DO elements_1
!-----------------------invert the preconditioner and get starting loads--
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                &
     value(fixed_freedoms),no(fixed_freedoms),store(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO  i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   diag_precon(no)=diag_precon(no)+penalty; loads(no)=diag_precon(no)*value
   store=diag_precon(no)
 END IF
 diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
 d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----------------------pcg equation solution-----------------------------
 pcg: DO
   cg_iters=cg_iters+1; u=zero
   elements_2: DO iel=1,nels
     g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
   END DO elements_2
   IF(fixed_freedoms/=0)u(no)=p(no)*store; up=DOT_PRODUCT(loads,d)
   alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha; loads=loads-u*alpha
   d=diag_precon*loads; beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
   CALL checon(xnew,x,cg_tol,cg_converged)
   IF(cg_converged.OR.cg_iters==cg_limit)EXIT
 END DO pcg
 WRITE(11,'(A,I5)')" Number of cg iterations to convergence was",cg_iters
 WRITE(11,'(/A)')"  Node   x-disp      y-disp      z-disp"; loads=xnew
 DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----------------------recover stresses at nip integrating point---------
 nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
 CALL sample(element,points,weights); loads(0)=zero
 WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
 WRITE(11,'(A,/,A)')"    Element     x-coord     y-coord     z-coord",    &
   "    sig_x       sig_y       sig_z       tau_xy      tau_yz      tau_zx"
 elements_3: DO iel=1,nels
```

```
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
   gauss_pts_2: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     sigma=MATMUL(dee,MATMUL(bee,eld)); WRITE(11,'(I8,4X,3E12.4)')iel,gc
     WRITE(11,'(6E12.4)')sigma
   END DO gauss_pts_2
 END DO elements_3
STOP
END PROGRAM p55
```

New scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling

New scalar reals:

`alpha`	α from equations (3.22)
`beta`	β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\{\mathbf{R}\}_k$ from equations (3.22)

Scalar logical:

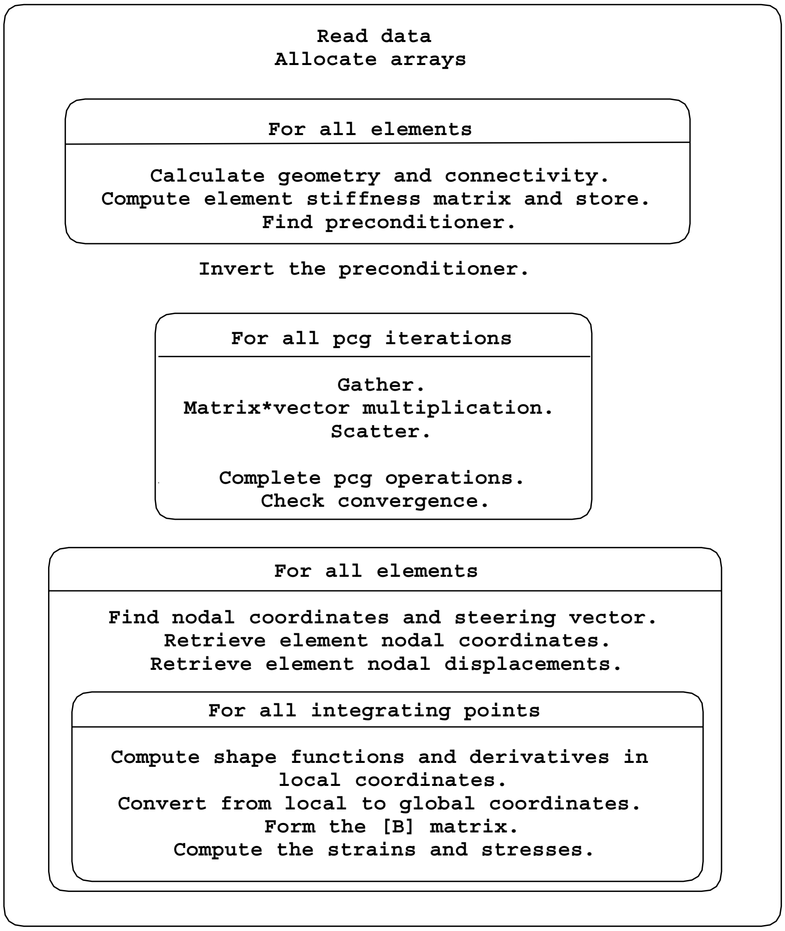
`cg_converged`	set to `.TRUE.` if pcg process has converged

New dynamic real arrays:

`d`	preconditioned rhs vector
`diag_precon`	diagonal preconditioner vector
`p`	"descent" vector used in equations (3.22)
`store`	stores augmented diagonal terms
`storkm`	holds element stiffness matrices
`u`	vector used in equations (3.22)
`x`	"old" solution vector
`xnew`	"new" solution vector

Section 3.5 described a "mesh-free" approach to the solution of linear static equilibrium problems in which the equation solution process could be carried out by the preconditioned conjugate gradient (pcg) technique without ever assembling element matrices into a global (stiffness) matrix. The essential process was described by equations (3.20) to (3.23). Program 5.5 will now be used to solve once more the problem illustrated in Figure 5.24 and previously solved using an assembly technique by Program 5.3. A structure chart for the pcg algorithm is shown in Figure 5.35.

All of the elements are looped in order to compute their stiffness matrices, which are stored in the array `storkm` for use later in the pcg solution algorithm. This loop (called



```
Read data
Allocate arrays
  For all elements
    Calculate geometry and connectivity.
    Compute element stiffness matrix and store.
    Find preconditioner.
Invert the preconditioner.
  For all pcg iterations
    Gather.
    Matrix*vector multiplication.
    Scatter.

    Complete pcg operations.
    Check convergence.
  For all elements
    Find nodal coordinates and steering vector.
    Retrieve element nodal coordinates.
    Retrieve element nodal displacements.
      For all integrating points
        Compute shape functions and derivatives in
        local coordinates.
        Convert from local to global coordinates.
        Form the [B] matrix.
        Compute the strains and stresses.
```

Figure 5.35 Structure chart for pcg algorithm as used in Program 5.5. No global matrix assembly

`elements_2`) also builds the preconditioning matrix which is simply the inverse of the diagonal terms in what would have been in the assembled global stiffness matrix. The preconditioning matrix (stored as a vector) is called `diag_precon`. The section commented "pcg equation solution" carries out the vector operations described in equations (3.22) within the iterative loop labelled `pcg`. The matrix–vector multiply needed in the first of (3.22) is done using (3.23). The steering vector `g` "gathers" the appropriate components of `p`, to be multiplied by the element stiffness matrix `km` retrieved from `storkm`. Similarly, the vector `g` "scatters" the result of the matrix–vector multiply to appropriate locations in `u`. A tolerance `pcg_tol` enables the iterations to be stopped when successive solutions

```
nod
20

nxe nye nze nip cg_tol cg_limit np_types
1    3   2   8  1.0e-5   200      2

prop(e,v)
100.0  0.3
 50.0  0.3

etype
1 2 1 2 1 2

x_coords, y_coords, z_coords
 0.0  0.5
 0.0  1.0  2.0  3.0
 0.0 -1.0 -2.0

nr,(k,nf(:,k),i=1,nr)
46
 1 0 0 1   2 1 0 1   3 1 0 1   4 0 0 1   5 1 0 1   6 0 0 1
 7 1 0 1   8 1 0 1   9 0 0 1  10 1 0 1  11 0 0 0  12 0 0 0
13 0 0 0  14 0 1 1  16 0 1 1  18 0 0 0  19 0 0 0  20 0 1 1
23 0 1 1  25 0 1 1  28 0 1 1  30 0 0 0  31 0 0 0  32 0 0 0
33 0 1 1  35 0 1 1  37 0 0 0  38 0 0 0  39 0 1 1  42 0 1 1
44 0 1 1  47 0 1 1  49 0 0 0  50 0 0 0  51 0 0 0  52 0 1 1
54 0 1 1  56 0 0 0  57 0 0 0  58 0 1 1  61 0 1 1  63 0 1 1
66 0 1 1  68 0 0 0  69 0 0 0  70 0 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
8
 1  0.0  0.0  0.0417    2  0.0  0.0 -0.1667    3  0.0  0.0  0.0417
14  0.0  0.0 -0.1667   15  0.0  0.0 -0.1667   20  0.0  0.0  0.0417
21  0.0  0.0 -0.1667   22  0.0  0.0  0.0417

fixed_freedoms
0
```

Figure 5.36 Data for Programs 5.5 and 5.6 example

are "close enough", but since `pcg` is a loop which might carry on "forever", an iteration ceiling, `cg_limit`, is specified also. Strains and stresses can then be recovered from the displacements in the usual manner.

The data for the program are shown in Figure 5.36. The only changes from Figure 5.24, which used an assembly strategy, are the two additional data values read in for `cg_tol` and `cg_limit`, which are set to be 1×10^{-5} and 200 respectively.

Figure 5.37 shows the results which may be compared with those listed in Figure 5.25. The specified accuracy of solution took 64 iterations in this case. Since, in perfect arithmetic, the conjugate gradient process should converge in at most `neq` iterations (124 in this case) the amount of computational effort in large problems may seem to be daunting. Fortunately, as the set of equations to be solved grows larger, the proportion of iterations to converge to number of equations (`cg_iters/neq`) usually decreases dramatically. It does however depend crucially on the "condition number" (nature of the eigenvalue spectrum) of the assembled stiffness matrix. With no diagonal preconditioning (`diag_precon=1.0`), the number of iterations for convergence rises to 70.

```
 There are  124 equations
 Number of cg iterations to convergence was   64

  Node   x-disp      y-disp      z-disp
    1  0.0000E+00  0.0000E+00 -0.2246E-01
    2  0.1584E-02  0.0000E+00 -0.2255E-01
    3  0.3220E-02  0.0000E+00 -0.2333E-01
    4  0.0000E+00  0.0000E+00 -0.1849E-01
    5  0.1544E-02  0.0000E+00 -0.1884E-01
    6  0.0000E+00  0.0000E+00 -0.1443E-01
    7  0.7580E-03  0.0000E+00 -0.1435E-01
    8  0.1511E-02  0.0000E+00 -0.1411E-01
    9  0.0000E+00  0.0000E+00 -0.6164E-02
   10  0.2792E-02  0.0000E+00 -0.6430E-02
.
.
.
   66  0.0000E+00  0.1572E-02 -0.1028E-03
   67 -0.7437E-04  0.1716E-02 -0.5845E-04
   68  0.0000E+00  0.0000E+00  0.0000E+00
   69  0.0000E+00  0.0000E+00  0.0000E+00
   70  0.0000E+00  0.0000E+00  0.0000E+00

 The integration point (nip= 1) stresses are:
    Element     x-coord     y-coord     z-coord
    sig_x       sig_y       sig_z       tau_xy      tau_yz      tau_zx
       1      0.2500E+00  0.5000E+00 -0.5000E+00
 -0.2671E-01 -0.1647E+00 -0.9088E+00  0.6148E-02  0.9598E-01  0.4352E-02
       2      0.2500E+00  0.5000E+00 -0.1500E+01
  0.3986E-01 -0.5316E-01 -0.6298E+00 -0.2139E-02  0.7613E-01  0.4169E-02
       3      0.2500E+00  0.1500E+01 -0.5000E+00
 -0.2481E-01 -0.1260E+00 -0.1052E+00  0.4842E-02  0.9398E-01 -0.2812E-02
       4      0.2500E+00  0.1500E+01 -0.1500E+01
  0.2477E-01 -0.8240E-01 -0.2822E+00 -0.3176E-02  0.1214E+00  0.2938E-02
       5      0.2500E+00  0.2500E+01 -0.5000E+00
  0.3756E-02  0.8610E-02 -0.1470E-01 -0.9130E-03 -0.8730E-02  0.8741E-03
       6      0.2500E+00  0.2500E+01 -0.1500E+01
  0.7401E-02 -0.4390E-01 -0.2832E-01 -0.5657E-03  0.6083E-01  0.4901E-04
```

Figure 5.37 Results from Programs 5.5 and 5.6 example

Program 5.6 Three-dimensional strain of an elastic solid using 8-, 14-, or 20-node brick hexahedra. Mesh numbered in *x*-*z* planes then in the *y*-direction. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver. Vectorised version.

```
PROGRAM p56
!-------------------------------------------------------------------------
! Program 5.6 Three-dimensional strain of an elastic solid using
!             8-, 14- or 20-node brick hexahedra. Mesh numbered in x-z
!             planes then in the y-direction. No global stiffness matrix
!             assembly. Diagonally preconditioned conjugate gradient solver.
!             Vectorised version.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,k,loaded_nodes,ndim=3,   &
   ndof,nels,neq,nip,nn,nprops=2,np_types,nod,nodof=3,nr,nst=6,nxe,nye,nze
 REAL(iwp)::alpha,beta,big,cg_tol,det,one=1.0_iwp,penalty=1.0e20_iwp,up,    &
   zero=0.0_iwp
 CHARACTER(LEN=15)::element='hexahedron'; LOGICAL::cg_converged
```

```
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),nf(:,:),no(:),    &
   node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),d(:),dee(:,:),der(:,:),       &
   deriv(:,:),diag_precon(:),eld(:),fun(:),gc(:),g_coord(:,:),g_pmul(:,:),&
   g_utemp(:,:),jac(:,:),km(:,:),loads(:),p(:),points(:,:),prop(:,:),     &
   sigma(:),store(:),storkm(:,:,:),u(:),value(:),weights(:),x(:),xnew(:), &
   x_coords(:),y_coords(:),z_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 read(10,*)nod,nxe,nye,nze,nip,cg_tol,cg_limit,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye,nze); ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),coord(nod,ndim),     &
   jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),fun(nod),gc(ndim),        &
   bee(nst,ndof),km(ndof,ndof),eld(ndof),sigma(nst),g_coord(ndim,nn),     &
   g_num(nod,nels),weights(nip),num(nod),g_g(ndof,nels),x_coords(nxe+1),  &
   g(ndof),y_coords(nye+1),z_coords(nze+1),etype(nels),g_pmul(ndof,nels), &
   prop(nprops,np_types),storkm(ndof,ndof,nels),g_utemp(ndof,nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords,z_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 WRITE(11,'(A,I5,A)')" There are",neq," equations"
 ALLOCATE(p(0:neq),loads(0:neq),x(0:neq),xnew(0:neq),u(0:neq),              &
   diag_precon(0:neq),d(0:neq))
 CALL sample(element,points,weights); diag_precon=zero
!----------element stiffness integration, storage and preconditioner------
 elements_1: DO iel=1,nels
   CALL hexahedron_xz(iel,x_coords,y_coords,z_coords,coord,num)
   CALL num_to_g(num,nf,g);   g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); g=g_g(:,iel); coord=TRANSPOSE(g_coord(:,num)); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1
   storkm(:,:,iel)=km
   DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
 END DO elements_1
!-----------------------invert the preconditioner and get starting loads--
 loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                   &
     value(fixed_freedoms),no(fixed_freedoms),store(fixed_freedoms))
   READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
   DO  i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   diag_precon(no)=diag_precon(no)+penalty; loads(no)=diag_precon(no)*value
   store=diag_precon(no)
 END IF
 diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
 d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----------------------pcg equation solution----------------------------
 pcg: DO
   cg_iters=cg_iters+1; u=zero
   elements_2: DO iel=1,nels; g_pmul(:,iel)=p(g_g(:,iel)); END DO elements_2
```

```
!dir$ ivdep
   elements_2a: DO iel=1,nels
     km=storkm(:,:,iel); g_utemp=MATMUL(km,g_pmul)
     u(g_g(:,iel))=u(g_g(:,iel))+g_utemp(:,iel)
   END DO elements_2a
   IF(fixed_freedoms/=0)u(no)=p(no)*store; up=DOT_PRODUCT(loads,d)
   alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha; loads=loads-u*alpha
   d=diag_precon*loads; beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
   big=zero; cg_converged=.TRUE.
   DO i=1,neq; IF(ABS(xnew(i))>big)big=ABS(xnew(i)); END DO
   DO i=1,neq; IF(ABS(xnew(i)-x(i))/big>cg_tol)cg_converged=.FALSE.; END DO
   x=xnew; IF(cg_converged.OR.cg_iters==cg_limit)EXIT
 END DO pcg
 WRITE(11,'(A,I5)')" Number of cg iterations to convergence was",cg_iters
 WRITE(11,'(/A)')"  Node   x-disp      y-disp      z-disp"; loads=xnew
 DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!----------------------recover stresses at nip integrating point---------
 nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
 CALL sample(element,points,weights); loads(0)=zero
 WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
 WRITE(11,'(A,/,A)')"    Element     x-coord     y-coord     z-coord",    &
   "    sig_x       sig_y       sig_z       tau_xy      tau_yz      tau_zx"
 elements_4: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
   gauss_pts_2: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     sigma=MATMUL(dee,MATMUL(bee,eld)); WRITE(11,'(I8,4X,3E12.4)')iel,gc
     WRITE(11,'(6E12.4)')sigma
   END DO gauss_pts_2
 END DO elements_4
STOP
END PROGRAM p56
```

New dynamic real arrays:

`g_pmul`	"gathers" the `p` vectors for all elements
`g_utemp`	holds all the products of `km` and `p`

This program is used to solve exactly the same problem as was detailed for Programs 5.3 and 5.5, using the mesh-free strategy of Program 5.5. However, it is used as an example of some of the issues which arise when programming for vector computers (see Section 1.4).

All practical vector computers enable code to be analysed to see where most time is being used and where there are features of the program inhibiting most effective use of vectorising compilers.

When Program 5.5 was run through such an analysis program, three main points arose:

a) There is potential "dependency" in the scatter operation

```
u(g)=u(g)+MATMUL(km,p(g))
```

and the compiler does not know whether there may be repeated entries in `g` and so does not vectorise this statement.

b) A surprising amount of time was spent in the Fortran 95 intrinsic `MAXVAL` (for testing convergence in subroutine `checon`).

c) The most time-consuming operation is the Fortran 95 intrinsic `MATMUL`, and on the particular vector computer, it was running considerably slower than the peak machine speed.

Program 5.6 addresses all of these issues. First, unless freedoms are "tied" together (a device not used in this book) we can be sure that entries in `g` are not duplicated and so the scatter operation can be vectorised. A "compiler directive" (`!dir$ ivdep` in this case) is therefore inserted before the loop `elements_2a:` enabling the loop to be vectorised. Second, `MAXVAL` is replaced by its longhand equivalent. This is obviously a problem with the particular vendor whose implementation of `MAXVAL` could be much improved. Third, and this is probably also a vendor problem, the `MATMUL` operation is changed from matrix–vector to matrix–matrix by collecting all the `p(g)` vectors into a global matrix `g_pmul` in the loop `elements_2:`. Otherwise the program is the same as Program 5.5 and of course produces the same results. However, Table 5.1 shows the progressive effects of making changes to the coding in Program 5.5.

Table 5.1 Timings of vectorised programs

Original code (Program 5.5)	44.7 seconds
No dependency	25.3 seconds
Replace `MAXVAL`	21.6 seconds
Matrix–matrix (Program 5.6)	9.3 seconds

The speed-up of Program 5.6 over Program 5.5 on this particular vector computer was by a factor of about 5, and illustrates the importance of code analysis when using such machines.

Glossary of variable names used in Chapter 5

Scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling
`fixed_freedoms`	number of fixed displacements
`i`	simple counter
`iel`	simple counter
`iflag`	1 for "symmetry", -1 for "antisymmetry"
`iwp`	`SELECTED_REAL_KIND(15)`
`k`	simple counter
`loaded_nodes`	number of loaded nodes
`lth`	harmonic on which loads are to be applied
`ndim`	number of dimensions
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh

`nip`	number of integrating points per element
`nn`	number of nodes in the mesh
`nod`	number of nodes per element
`nodof`	number of degrees of freedom per node
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes
`nre`	number of elements in r-direction
`nst`	number of stress (strain) terms (3, 4, or 6)
`nxe`	number of elements in x-direction
`nye`	number of elements in y-direction
`nze`	number of elements in z-direction

Scalar reals:

`alpha`	α from equations (3.22)
`beta`	β from equations (3.22)
`ca`	set to `cos(chi)`
`chi`	angle for stress output
`cg_tol`	pcg convergence tolerance
`det`	determinant of the Jacobian matrix
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`pi`	set to π
`radius`	r-coordinate of Gauss point
`sa`	set to `sin(chi)`
`up`	holds dot product $(\mathbf{R}^k)^{\mathrm{T}}(\mathbf{R}^k)$ from equations (3.22)
`zero`	set to 0.0

Scalar characters:

`dir`	element and node numbering direction
`element`	element type
`type_2d`	type of 2D analysis (`'plane'` or `'axisymmetric'`)

Scalar logical:

`cg_converged`	set to `.TRUE.` if pcg process has converged

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`kdiag`	diagonal term location vector
`nf`	nodal freedom matrix
`no`	fixed freedom numbers vector
`node`	fixed nodes vector

`num`	element node numbers vector
`sense`	sense of freedoms to be fixed vector

Dynamic real arrays:

`bee`	strain-displacement matrix
`coord`	element nodal coordinates
`d`	preconditioned rhs vector
`dee`	stress–strain matrix
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`diag_precon`	diagonal preconditioner vector
`eld`	element nodal displacements
`fun`	shape functions
`gc`	integrating point coordinates
`gravlo`	loads generated by gravity
`g_coord`	global nodal coordinates
`g_pmul`	"gathers" the `p` vectors for all elements
`g_utemp`	holds all the products of `km` and `p`
`jac`	Jacobian matrix
`km`	element stiffness matrix
`kv`	global stiffness matrix
`loads`	nodal loads and displacements
`p`	"descent" vector used in equations (3.22)
`points`	integrating point local coordinates
`prop`	element properties
`r_coords`	r-coordinates of mesh layout
`sigma`	stress terms
`store`	stores augmented diagonal terms
`storkm`	holds element stiffness matrices
`u`	vector used in equations (3.22)
`value`	fixed values of displacements
`weights`	weighting coefficients
`x`	"old" solution vector
`xnew`	"new" solution vector
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout
`z_coords`	z-coordinates of mesh layout

5.2 Exercises

1. Derive in terms of local coordinates, any shape function of the following elements:
   (a) 6-node triangle
   (b) 8-node quadrilateral
   (c) 9-node quadrilateral

2. Given that the "first" shape function of a 4-node plane stress rectangular element of width a and height b is:

$$N_1 = \left(1 - \frac{x}{a}\right)\left(1 - \frac{y}{b}\right)$$

use analytical integration to show that the element stiffness and mass matrices include the following terms:

$$k_{11} = \frac{E}{1-\nu^2}\left(\frac{b}{3a} + \frac{1-\nu}{2}\frac{a}{3b}\right)$$

$$m_{11} = \frac{\rho ab}{9}$$

3. For the problem shown in Figure 5.38, estimate the force necessary to displace the loaded node horizontally by 0.015 units.

Ans: 0.8

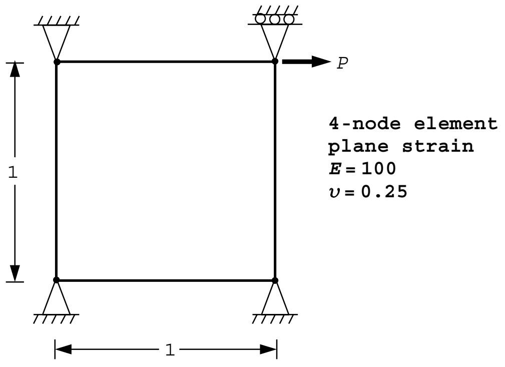


Figure 5.38

4. Derive the vertical nodal forces that are equivalent to the triangular stress distribution acting on the 4-node element shown in Figure 5.39. Given that the stiffness matrix of this element (assuming local freedom numbering in the order u_1 v_1 u_2 v_2 u_3 v_3 u_4 v_4) is:

$$\begin{bmatrix} 57.69 & 24.04 & 9.62 & -4.81 & -28.85 & -24.04 & -38.46 & 4.81 \\ & 57.69 & 4.81 & -38.46 & -24.04 & -28.85 & -4.81 & 9.62 \\ & & 57.69 & -24.04 & -38.46 & -4.81 & -28.85 & 24.04 \\ & & & 57.69 & 4.81 & 9.62 & 24.04 & -28.85 \\ & & & & 57.69 & 24.04 & 9.62 & -4.81 \\ & & & & & 57.69 & 4.81 & -38.46 \\ & & & & & & 57.69 & -24.04 \\ & & & & & & & 57.69 \end{bmatrix}$$

compute the vertical displacement of the top two nodes.

Ans: 0.55, 0.19

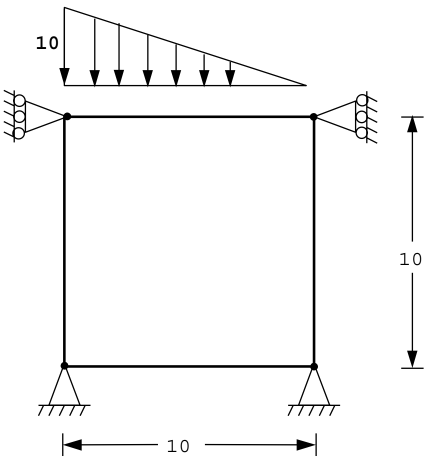


Figure 5.39

5. Derive the equivalent nodal force F_5 that is one of the equivalent nodal forces required to model a linear triangular distributed load varying from one to zero as shown in Figure 5.40. Given that $F_4 = L/3$ what must F_3 be equal to?

   Ans: $F_3 = 0,\ F_4 = L/3,\ F_5 = L/6$

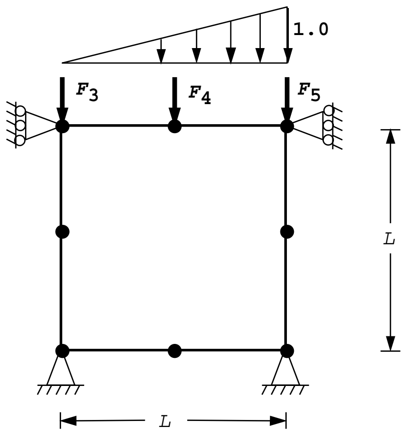


Figure 5.40

6. The 4-node element shown in Figure 5.41 is fixed on three sides and subjected to the indicated fixed displacement at the free node. Estimate the stresses at the centroid of the element assuming plane strain conditions.

   Ans: $$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \begin{Bmatrix} 6.66 \\ 4.96 \\ -3.50 \end{Bmatrix}$$

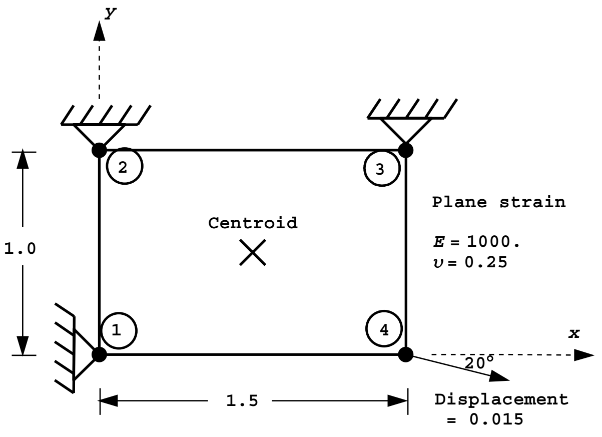


Figure 5.41

7. The rectangular 8-node element shown in Figure 5.42 has a unit weight of γ and is subjected to gravitational loading. Compute the equivalent vertical nodal load F_{corner} at node 1.
   If it can be assumed that the equivalent loads due to gravity result in all corner loads equalling F_{corner} and all mid-side loads equaling $F_{\text{mid-side}}$, deduce also the value of $F_{\text{mid-side}}$.
   (Ans: $F_{\text{corner}} = ab\gamma/12$, $F_{\text{mid-side}} = -ab\gamma/3$)

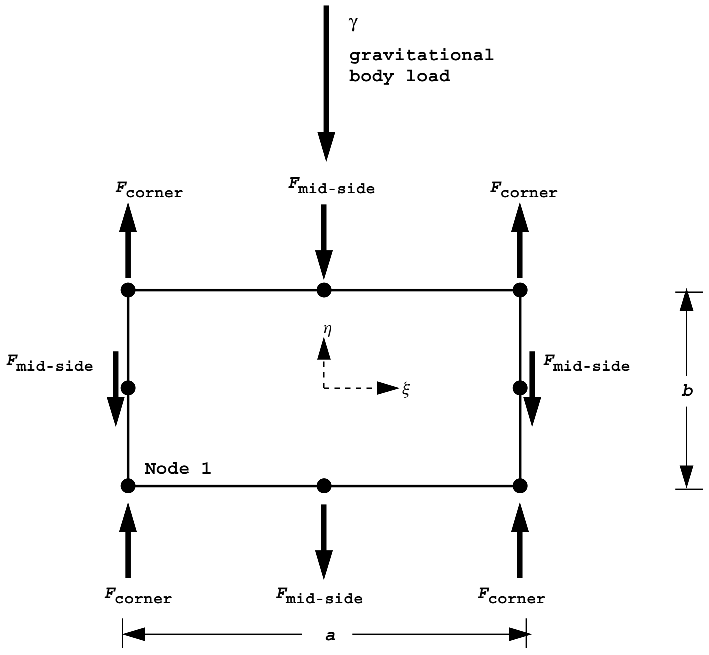


Figure 5.42

8. The global stiffness matrix for the problem shown in Figure 5.43 is given by

$$[\mathbf{K}]_m = \begin{bmatrix} \alpha & 12.5 \\ 12.5 & 75 \end{bmatrix}$$

Derive the missing term α and hence compute the displacement of the loaded node.

Ans: $\alpha = 50$, $\delta_x = -0.017$, $\delta_y = -0.010$

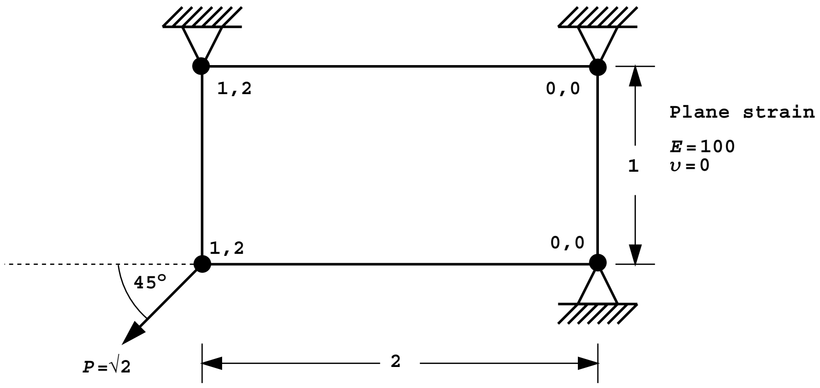


Figure 5.43

9. Compute the equivalent nodal loads for the uniform distributed loading applied over one half of one side of the 8-node quadrilateral element shown in Figure 5.44.

Ans: $F_5 = -6$, $F_6 = 48$, $F_7 = 30$,

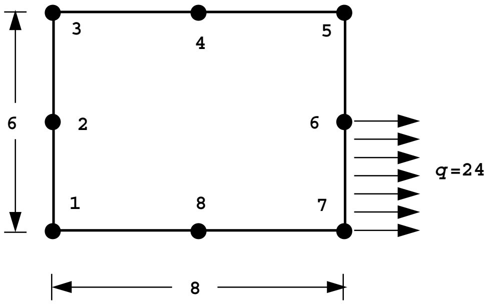


Figure 5.44

10. The third member of the triangular family has 10 nodes as shown in Figure 5.45. Set up the system of simultaneous equations that would enable you to derive the shape function N_{10} in local coordinates for this element. Do not attempt to solve the equations.

Ans: $N_{10} = c_1 + c_2L_1 + c_3L_2 + c_4L_1^2 + c_5L_1L_2 + c_6L_2^2 + c_7L_1^3 + c_8L_1^2L_2$
$+c_9L_1L_2^2 + c_{10}L_2^3$

$$\begin{bmatrix} 27 & 27 & 0 & 27 & 0 & 0 & 27 & 0 & 0 & 0 \\ 27 & 0 & 27 & 0 & 0 & 27 & 0 & 0 & 0 & 27 \\ 27 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 27 & 18 & 9 & 12 & 6 & 3 & 8 & 4 & 2 & 1 \\ 27 & 9 & 18 & 3 & 6 & 12 & 1 & 2 & 4 & 8 \\ 27 & 0 & 18 & 0 & 0 & 12 & 0 & 0 & 0 & 8 \\ 27 & 0 & 9 & 0 & 0 & 3 & 0 & 0 & 0 & 1 \\ 27 & 9 & 0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 \\ 27 & 18 & 0 & 12 & 0 & 0 & 8 & 0 & 0 & 0 \\ 27 & 9 & 9 & 3 & 3 & 3 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \\ c_{10} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 27 \end{Bmatrix}$$

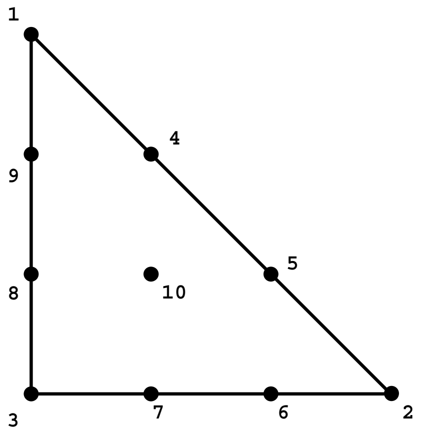


Figure 5.45

11. Figure 5.46 shows a cubic, 8-node, 3D finite element with side length 2 units. The origin of the coordinate system is at the centroid of the element, so all nodal coordinates are ± 1 (e.g. node 1 is at $(-1, -1, -1)$ etc.). Choose suitable terms for the shape functions of this element, and hence set up the system of simultaneous equations that would enable you to derive shape function N_7. Do not attempt to solve the equations.

Ans: $N_7 = c_1 + c_2x + c_3y + c_4z + c_5xy + c_6yz + c_7zx + c_8xyz$

$$\begin{bmatrix} 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{Bmatrix}$$

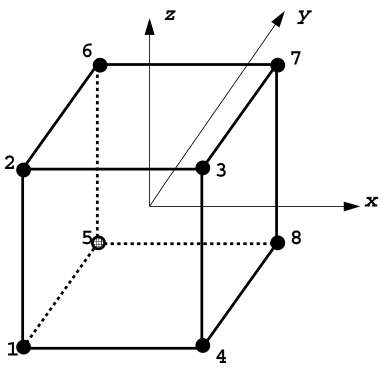


Figure 5.46

12. Assuming "small" strains, derive the partial differential equations of 2D elastic equilibrium under conditions of plane stress. For the square element shown in Figure 5.47 show that the stiffness matrix contains the term,

$$k_{12} = \frac{E\nu}{6(1-\nu^2)}$$

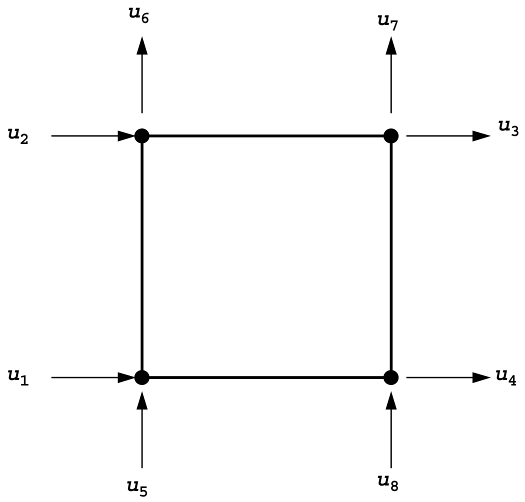


Figure 5.47

13. Selective reduced integration (SRI) is a way of improving the performance of 4-node plane elements as Poisson's ratio approaches 0.5 (incompressibility). The [**D**] matrix is split into volumetric and deviatoric components, and the element stiffness matrix

is integrated in two stages, namely

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}]^{v}[\mathbf{B}]\,\mathrm{d}x\,\mathrm{d}y + \iint [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}]^{d}[\mathbf{B}]\,\mathrm{d}x\,\mathrm{d}y$$

where

$$[\mathbf{D}]^d = \frac{E}{2(1+\nu)}\begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and

$$[\mathbf{D}]^v = \frac{E\nu}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The SRI approach involves "exact" integration (`nip = 4`) of the $[\mathbf{D}]^d$ term, and "reduced" integration (`nip = 1`) of the $[\mathbf{D}]^v$ term.

Use this technique to estimate the displacement of the loaded node in Figure 5.48. You may use analytical integration for the $[\mathbf{D}]^d$ term.

Ans: $\delta_x = 0.2$, $\delta_x = -0.2$

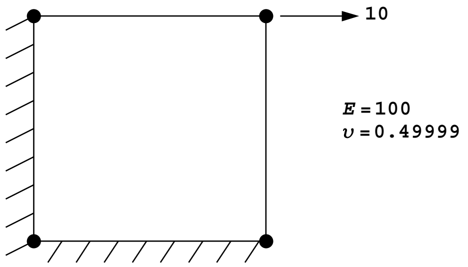


Figure 5.48

14. A planar square 8-node quadrilateral of unit side length and unit mass density has the following shape functions at node 1:

$$N_1 = 0.25(1-\xi)(1-\eta)(-\eta-\xi-1)$$

Compute m_{11} of the element consistent mass matrix using (a) `nip=4` and (b) `nip=9`

Ans: (a) $m_{11} = 0.0185$, (b) $m_{11} = 0.0333$

15. Use Program 5.1 to compute the vertical deflection at the edge of the flexible footing shown in Figure 5.49. You should use symmetry to reduce the number of elements in the discretisation.

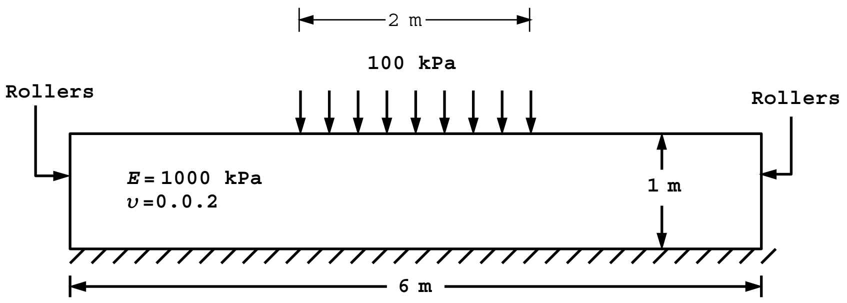


Figure 5.49

References

Cuthill E and McKee J 1969 Reducing the bandwidth of sparse symmetric matrices. *ACM Proceedings of the 24th National Conference*, New York.

Griffiths DV 1986 HARMONY - A program for predicting the response of axisymmetric bodies subjected to non-axisymmetric loading. Technical Report GRC-96-44, Arthur Lakes Library, Colorado School of Mines, Geomechanics Research Center.

Hicks MA and Mar A 1996 A benchmark computational study of finite element error estimation. *Int J Numer Methods Eng* **39**(23), 3969–3983.

Irons BM 1971 Quadrature rules for brick-based finite elements. *Int J Numer Methods Eng* **3**, 293–294.

Peano A 1987 Inadmissible distortion of solid elements and patch tests results. *Commun Appl Numer Methods* **5**, 97–101.

Poulos HG and Davis EH 1974 *Elastic Solutions for Soil and Rock Mechanics*. John Wiley & Sons, Chichester, New York.

Smith IM and Kidger DJ 1992 Elastoplastic analysis using the 14-node brick element family. *Int J Numer Methods Eng* **35**, 1263–1275.

Wilson EL 1965 Structural analysis of axisymmetric solids. *J Am Inst Aeronaut Astronaut* **3**, 2269–2274.

Zienkiewicz OC and Taylor RL 1989 *The Finite Element Method*, vol. 1, 4th edn. McGraw-Hill, London, New York.

6

Material Non-linearity

6.1 Introduction

Non-linear processes pose very much greater analytical problems than do the linear processes so far considered in this book. The non-linearity may be found in the dependence of the equation coefficients on the solution itself or in the appearance of powers and products of the unknowns or their derivatives.

Two main types of non-linearity can manifest themselves in finite element analysis of solids: material non-linearity, in which the relationship between stresses and strains (or other material properties) are complicated functions, which result in the equation coefficients depending on the solution, and geometric non-linearity (otherwise known as "large strain" or "large displacement" analysis), which leads to products of the unknowns in the equations.

In order to keep the present book to a manageable size, the 11 programs described in this chapter deal only with material non-linearity. As far as the organisation of computer programs is concerned, material non-linearity is simpler to implement than geometric non-linearity. However, readers will appreciate how programs could be adapted to cope with geometric non-linearity as well (see e.g. Smith, 1997).

In practical finite element analysis two main types of solution procedure can be adopted to model material non-linearity. The first approach, which has already been seen in Program 4.5, involves "constant stiffness" iterations in which non-linearity is introduced by iteratively modifying the right hand side "loads" vector. The (usually elastic) global stiffness matrix in such an analysis is formed once only. Each iteration thus represents an elastic analysis of the type described in Chapter 5. Convergence is said to occur when stresses generated by the loads satisfy some stress–strain law or yield or failure criterion within prescribed tolerances. The loads vector at each iteration consists of externally applied loads and self-equilibrating "body loads". The body loads have the effect of redistributing stresses (or moments) within the system, but as they are self-equilibrating, they do not alter the net loading on the system. The "constant stiffness" method is shown diagrammatically in Figure 6.1. For load-controlled problems, many iterations may be required as

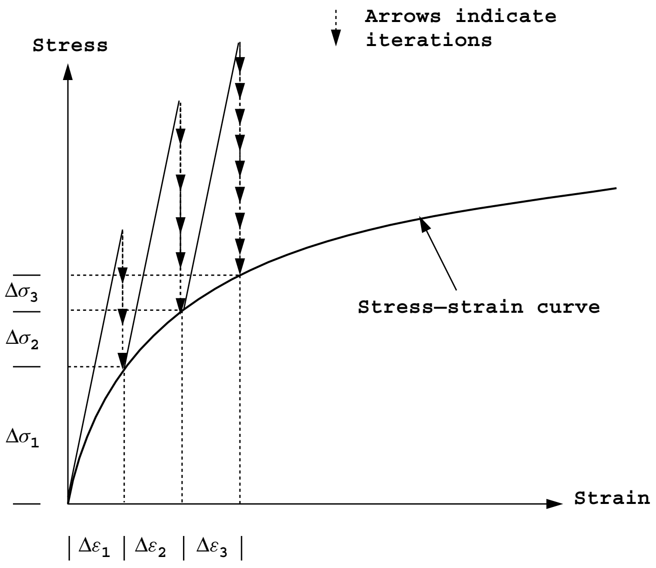


Figure 6.1 Constant stiffness method

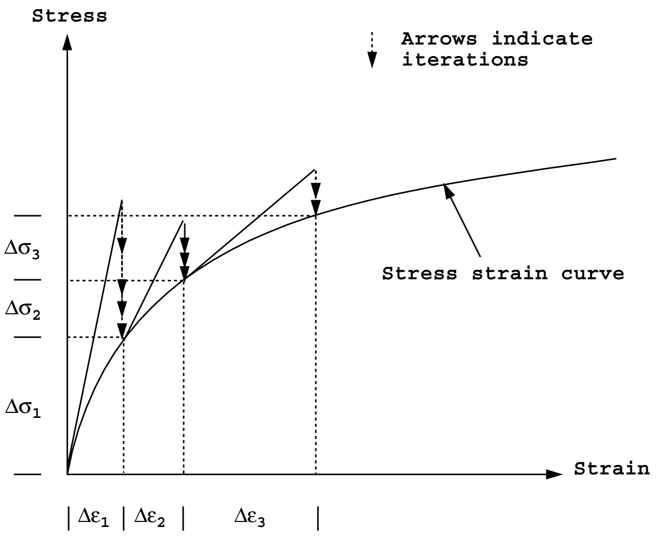


Figure 6.2 Variable (tangent) stiffness method

failure is approached, because the elastic (constant) global stiffness matrix starts to seriously overestimate the actual material stiffness.

Less iterations per load step are required if the second approach, the "variable" or "tangent" stiffness method is adopted. This method, shown in Figure 6.2, takes account of the reduction in stiffness of the material as failure is approached. If small enough load steps are taken, the method can become equivalent to a simple Euler "explicit" method. In practice, the global stiffness matrix may be updated periodically and "body loads" iterations employed to achieve convergence. In contrasting the two methods, the extra cost of reforming and factorising the global stiffness matrix in the "variable stiffness" method is offset by reduced numbers of iterations, especially as failure is approached.

A further possibility, introduced in later programs, is "implicit" integration of the rate equations rather than the "explicit" methods just described. This helps to further reduce the number of iterations to convergence.

Programs 6.1 to 6.4 employ the "constant stiffness" approach and explicit integration, and are similar in structure to Program 4.5 described previously for plastic analysis of frames. Both the viscoplastic method and the simple initial stress method are implemented. Programs 6.5 and 6.6 introduce tangent stiffness algorithms, and Programs 6.7 and 6.8 describe procedures for embanking and excavation in which construction sequences can be realistically modelled. Program 6.9 introduces a simple technique for modelling pore pressures in a saturated soil, and Programs 6.10 and 6.11 complete the chapter with 3D elasto-plastic analyses of slopes. Mesh-free solution strategies are described in Programs 6.2, 6.6, and 6.11. Many of the examples are chosen because they have closed-form solutions for comparison.

Before describing the programs, some discussion is necessary regarding the form of the stress–strain laws that are to be adopted. In addition, two popular methods of generating body loads for "constant stiffness" methods, namely "viscoplasticity" and "initial stress" are described.

6.2 Stress–strain behaviour

Although non-linear elastic constitutive relations have been applied in finite element analyses and especially soil mechanics applications (e.g. Duncan and Chang, 1970), the main physical feature of non-linear material behaviour is usually the irrecoverability of strain. A convenient mathematical framework for describing this phenomenon is to be found in the theory of plasticity (e.g. Hill, 1950). The simplest stress–strain law of this type that could be implemented in a finite element analysis involves elastic-perfectly plastic material behaviour (Figure 6.3). Although a simple law of this type was described in Chapter 4 (Figure 4.28), it is convenient in solid mechanics to introduce a "yield" surface in principal stress space which separates stress states that give rise to elastic and to plastic (irrecoverable) strains. To take account of complicated processes like cyclic loading, the yield surface may move in stress space "kinematically" (e.g. Molenkamp, 1987) but in this book only immovable surfaces are considered. An additional simplification introduced here is that the yield and ultimate "failure" surfaces are identical.

Algebraically, the surfaces are expressed in terms of a yield or failure function F. This function, which has units of stress, depends on the material strength and invariant combinations of the stress components. The function is designed such that it is negative within the yield or failure surface and zero on the yield or failure surface. Positive values of F imply stresses lying outside the yield or failure surface which are illegal and which must be redistributed via the iterative process described previously.

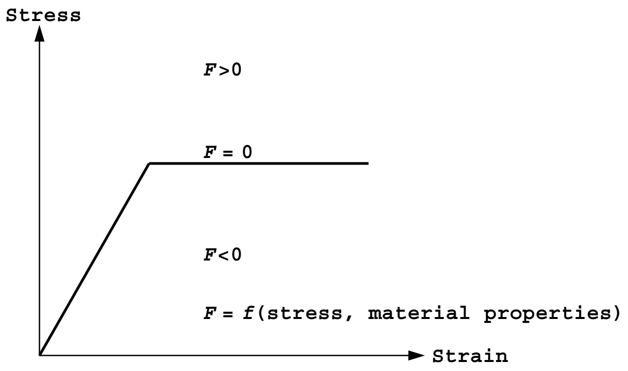


Figure 6.3 Elastic-perfectly plastic stress–strain law

During plastic straining, the material may flow in an "associated" manner, that is the vector of plastic strain increment may be normal to the yield or failure surface. Alternatively, normality may not exist and the flow may be "non-associated". Associated flow leads to various mathematically attractive simplifications and, when allied to the von Mises or Tresca failure criterion, correctly predicts zero plastic volume change during yield for undrained clays. For frictional materials, whose ultimate state is described by the Mohr–Coulomb criterion, associated flow leads to physically unrealistic volumetric expansion or dilation during yield. In such cases, non-associated flow rules are preferred in which plastic straining is described by a plastic potential function Q. This function may be geometrically similar to the failure function F but with the friction angle ϕ replaced by a dilation angle ψ. The implementation of the plastic potential function will be described further in Sections 6.6 and 6.7.

Before outlining some commonly used failure criteria and their representations in principal stress space, some useful stress invariant expressions are reviewed briefly.

6.3 Stress invariants

The Cartesian stress tensor defining the stress conditions at a point within a loaded body is given by:

$$\left[\, \sigma_x \; \sigma_y \; \sigma_z \; \tau_{xy} \; \tau_{yz} \; \tau_{zx} \,\right]^{\mathrm{T}} \tag{6.1}$$

which can be shown to be equivalent to three principal stresses acting on orthogonal planes:

$$[\, \sigma_1 \; \sigma_2 \; \sigma_3 \,]^{\mathrm{T}} \tag{6.2}$$

Principal stress space is obtained by treating the principal stresses as three-dimensional coordinates and is a useful way of representing a stress state at a point. It may be noted that although principal stress space defines the magnitudes of the principal stresses, it gives no indication of their orientation in physical space.

Instead of defining a point in principal stress space with coordinates $(\sigma_1,\ \sigma_2,\ \sigma_3)$ it is often more convenient to use invariants $(s,\ t,\ \theta)$, defined as:

$$
\begin{aligned}
s &= \frac{1}{\sqrt{3}}\left(\sigma_x + \sigma_y + \sigma_z\right) \\
t &= \frac{1}{\sqrt{3}}\left[(\sigma_x - \sigma_y)^2 + (\sigma_y - \sigma_z)^2 + (\sigma_z - \sigma_x)^2 + 6\tau_{xy}^2 + 6\tau_{yz}^2 + 6\tau_{zx}^2\right]^{1/2} \\
\theta &= \frac{1}{3}\arcsin\left(\frac{-3\sqrt{6}J_3}{t^3}\right)
\end{aligned}
\tag{6.3}
$$

where

$$J_3 = s_x s_y s_z - s_x\tau_{yz}^2 - s_y\tau_{zx}^2 - s_z\tau_{xy}^2 + 2\tau_{xy}\tau_{yz}\tau_{zx}$$

and

$$s_x = \frac{(2\sigma_x - \sigma_y - \sigma_z)}{3} \quad \text{etc.}$$

These expressions assume a compression-negative sign convention.

As shown in Figure 6.4, s gives the distance from the origin to the π-plane in which the stress point lies, and t represents the perpendicular distance of the stress point from the space diagonal. The Lode angle θ (called `lode_theta` in the programs) is a measure of the angular position of the stress point within the π-plane.

It may be noted that in some geotechnical applications, plane strain conditions apply and equations (6.3) are simplified because $\tau_{yz} = \tau_{zx} = 0$.

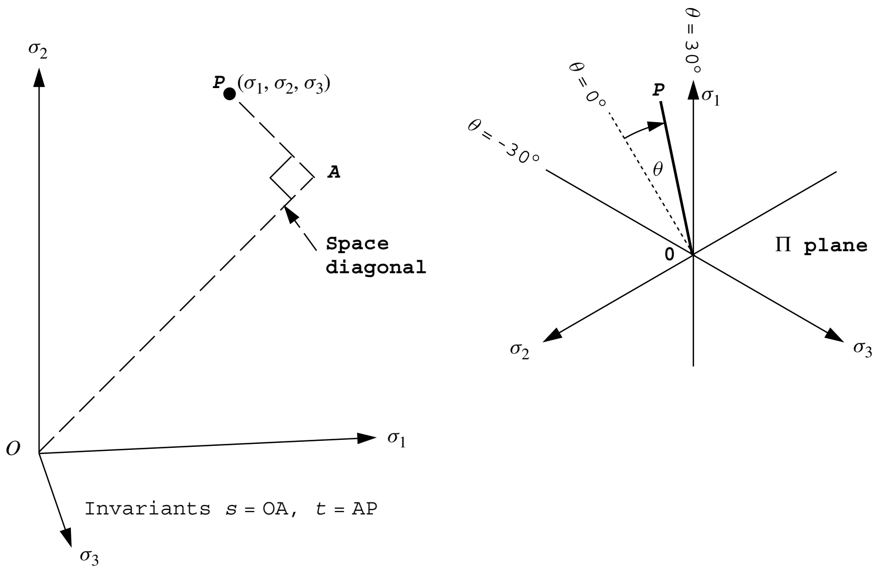


Figure 6.4 Representation of stress in principal stress space

In the programs described later in this chapter, the invariants that are used are slightly different to those defined in (6.3) whence:

$$\sigma_m = \frac{1}{\sqrt{3}}\, s$$
$$\overline{\sigma} = \sqrt{\frac{3}{2}}\, t \tag{6.4}$$

These expressions, called `sigm` and `dsbar` in program terminology, have more physical meaning than s and t in that they represent respectively the "mean stress" and "deviator stress" in a triaxial test. The relationship between principal stresses and invariants is given as follows:

$$\sigma_1 = \sigma_m + \frac{2}{3}\,\overline{\sigma}\sin\left(\theta - \frac{2\pi}{3}\right)$$
$$\sigma_2 = \sigma_m + \frac{2}{3}\,\overline{\sigma}\sin\theta \tag{6.5}$$
$$\sigma_3 = \sigma_m + \frac{2}{3}\,\overline{\sigma}\sin\left(\theta + \frac{2\pi}{3}\right)$$

which ensures that σ_1 is the most compressive and σ_3 is the least compressive. The Lode angle θ varies in the range $-\pi/6 \le \theta \le \pi/6$ ($-30^\circ \le \theta \le 30^\circ$), where $\theta = 30^\circ$ corresponds to "triaxial compression" ($\sigma_2 = \sigma_3$), and $\theta = -30^\circ$ corresponds to "triaxial extension" ($\sigma_1 = \sigma_2$).

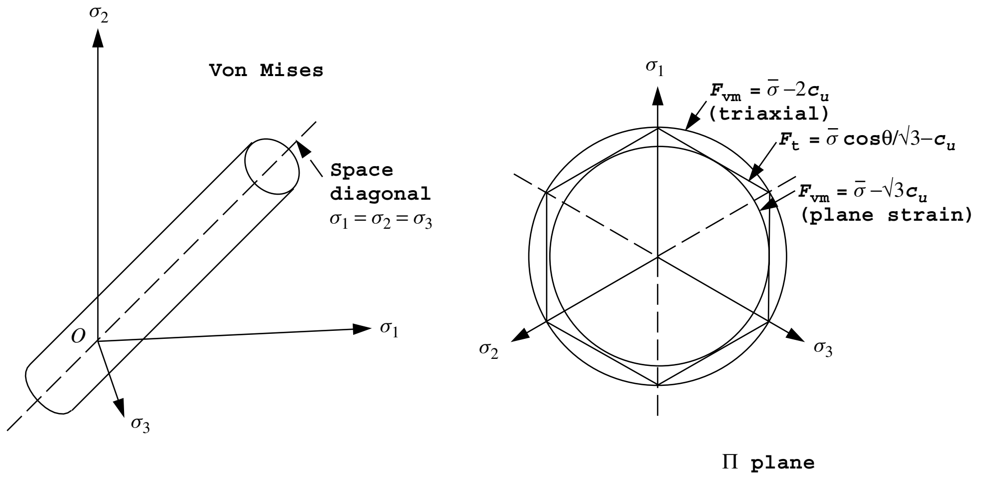
6.4 Failure criteria

Several failure criteria have been proposed for representing the strength of soils as engineering materials. For soils with both frictional and cohesive components of shear strength, conical failure criteria are appropriate, the best known of which is undoubtedly the Mohr–Coulomb criterion. For metals or undrained clays which behave in a "frictionless" ($\phi_u = 0$) manner, cylindrical failure criteria are appropriate and are discussed first.

6.4.1 Von Mises

As shown in Figure 6.5(a), this criterion takes the form of a right circular cylinder lying along the space diagonal. Only one of the three invariants, namely t (or $\overline{\sigma}$), is of any significance when determining whether a stress state has reached the limit of elastic behaviour. The onset of yield in a von Mises material is not dependent upon invariants s or θ.

The symmetry of the von Mises criterion when viewed in the π-plane indicates why it is not ideally suited to correlations with traditional soil mechanics concepts of strength. The criterion gives equal weighting to all three principal stresses, so if it is to be used to model undrained clay behaviour, consideration must be given to the value of the intermediate principal stress, σ_2, at failure.



σ_2
Von Mises
Space diagonal
$\sigma_1 = \sigma_2 = \sigma_3$
O
σ_1
σ_3
σ_1
$F_{vm} = \bar{\sigma} - 2c_u$ (triaxial)
$F_t = \bar{\sigma}\cos\theta/\sqrt{3} - c_u$
$F_{vm} = \bar{\sigma} - \sqrt{3}c_u$ (plane strain)
σ_2
σ_3
Π plane

Figure 6.5 Von Mises and Tresca failure criteria

For plane strain applications assuming no plastic volume change, it can be shown that at failure

$$\sigma_2 = \frac{\sigma_1 + \sigma_3}{2} \tag{6.6}$$

hence the von Mises criterion given by:

$$F_{\text{vm}} = \overline{\sigma} - \sqrt{3}c_u \tag{6.7}$$

should be used, where c_u is the undrained "cohesion" or shear strength of the soil.

On the other hand, under triaxial conditions, where:

$$\sigma_2 = \sigma_3 \tag{6.8}$$

the required von Mises criterion is given by

$$F_{\text{vm}} = \overline{\sigma} - 2c_u \tag{6.9}$$

Both of these expressions when applied to the appropriate stress condition, ensure that at failure,

$$\left|\frac{\sigma_1 - \sigma_3}{2}\right| = c_u \tag{6.10}$$

6.4.2 Mohr–Coulomb and Tresca

In principal stress space, this criterion takes the form of an irregular hexagonal cone, as shown in Figure 6.6. The irregularity is due to the fact that σ_2 is not taken into account. In order to derive the invariant form of this criterion, it should first be written in terms

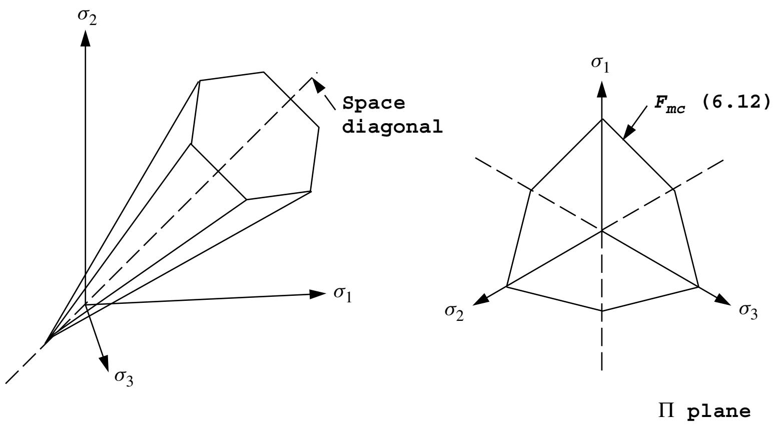


Figure 6.6 Mohr–Coulomb failure criterion

of principal stresses from the geometry of Mohr's circle, thus (assuming compression-negative):

$$F_{\mathrm{mc}} = \frac{\sigma_1 + \sigma_3}{2}\sin\phi - \frac{\sigma_1 - \sigma_3}{2} - c\cos\phi \tag{6.11}$$

Substituting for σ_1 and σ_3 from (6.5) gives the function:

$$F_{\mathrm{mc}} = \sigma_m \sin\phi + \overline{\sigma}\left(\frac{\cos\theta}{\sqrt{3}} - \frac{\sin\theta\sin\phi}{3}\right) - c\cos\phi \tag{6.12}$$

which shows that the Mohr–Coulomb criterion depends on all three invariants (σ_m, $\overline{\sigma}$, θ).

The Tresca criterion is obtained from (6.12) by putting $\phi_u = 0$ to give:

$$F_{\mathrm{t}} = \frac{\overline{\sigma}\cos\theta}{\sqrt{3}} - c_u \tag{6.13}$$

This criterion is preferred to von Mises for applications involving undrained clays because (6.10) is always satisfied at failure, regardless of the value of σ_2. In principal stress space the Tresca criterion is a regular hexagonal cylinder tangential to the von Mises cylinder defined by (6.7) and circumscribed by the one defined in (6.9) as shown in Figure 6.5.

6.5 Generation of body loads

Constant stiffness methods of the type described in this chapter use repeated elastic solutions to achieve convergence by iteratively varying the loads on the system. Within each load increment, the system of equations

$$[\mathbf{K}_m]\{\mathbf{U}\}^i = \{\mathbf{F}\}^i \tag{6.14}$$

must be solved for the global displacement increments $\{\mathbf{U}\}^i$, where i represents the iteration number, $[\mathbf{K}_m]$ the global stiffness matrix, and $\{\mathbf{F}\}^i$ the global external and internal (body)loads.

The element displacement increments $\{\mathbf{u}\}^i$ are extracted from $\{\mathbf{U}\}^i$, and these lead to strain increments via the element strain-displacement relationships:

$$\{\boldsymbol{\Delta\epsilon}\}^i = [\mathbf{B}]\{\mathbf{u}\}^i \tag{6.15}$$

Assuming the material is yielding, the strains will contain both elastic and (visco) plastic components, thus

$$\{\boldsymbol{\Delta\epsilon}\}^i = \left\{\boldsymbol{\Delta\epsilon}^e\right\}^i + \left\{\boldsymbol{\Delta\epsilon}^p\right\}^i \tag{6.16}$$

It is only the elastic strain increments $\left\{\boldsymbol{\Delta\epsilon}^e\right\}^i$ that generate stresses through the elastic stress–strain matrix, hence

$$\{\boldsymbol{\Delta\sigma}\}^i = [\mathbf{D}^e]\left\{\boldsymbol{\Delta\epsilon}^e\right\}^i \tag{6.17}$$

These stress increments are added to stresses already existing from the previous load step and the updated stresses substituted into the failure criterion (e.g. 6.12). If stress redistribution is necessary ($F > 0$), this is done by altering the load increment vector $\{\mathbf{F}\}^i$ in equation (6.14). In general, this vector holds two types of load, as given by

$$\{\mathbf{F}\}^i = \{\mathbf{F}_a\} + \{\mathbf{F}_b\}^i \tag{6.18}$$

where $\{\mathbf{F}_a\}$ is the actual applied external load increment and $\{\mathbf{F}_b\}^i$ is the body loads vector that varies from one iteration to the next. The $\{\mathbf{F}_b\}^i$ vector must be self-equilibrating so that the net loading on the system is not affected by it. Two simple methods for generating body loads are now described briefly.

6.6 Viscoplasticity

In this method (Zienkiewicz and Cormeau, 1974) , the material is allowed to sustain stresses outside the failure criterion for finite "periods". Overshoot of the failure criterion, as signified by a positive value of F, is an integral part of the method and is actually used to drive the algorithm.

Instead of plastic strains, we now refer to viscoplastic strains and these are generated at a rate that is related to the amount by which yield has been violated through the expression

$$\left\{\dot{\boldsymbol{\epsilon}}^{vp}\right\} = F\left\{\frac{\partial Q}{\partial \sigma}\right\} \tag{6.19}$$

where F is the yield function and Q is the plastic potential function.

It should be noted that a pseudo-viscosity property equal to unity is implied on the right hand side of equation (6.19) from dimensional considerations.

Multiplication of the viscoplastic strain rate by a pseudo-time step gives an increment of viscoplastic strain which is accumulated from one “time step” or iteration to the next; thus

$$\left\{\boldsymbol{\delta\epsilon}^{vp}\right\}^i = \Delta t \left\{\dot{\boldsymbol{\epsilon}}^{vp}\right\}^i \tag{6.20}$$

and

$$\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}^i = \left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}^{i-1} + \left\{\boldsymbol{\delta\epsilon}^{vp}\right\}^i \tag{6.21}$$

The “time step” for unconditional numerical stability has been derived by Cormeau (1975) and depends on the assumed failure criterion. Thus, for von Mises materials:

$$\Delta t = \frac{4(1+\nu)}{3E} \tag{6.22}$$

and for Mohr–Coulomb materials:

$$\Delta t = \frac{4(1+\nu)(1-2\nu)}{E(1-2\nu+\sin^2\phi)} \tag{6.23}$$

The derivatives of the plastic potential function Q with respect to stresses are conveniently expressed through the Chain Rule, thus

$$\left\{\frac{\partial Q}{\partial \boldsymbol{\sigma}}\right\} = \frac{\partial Q}{\partial \sigma_m}\left\{\frac{\partial \sigma_m}{\partial \boldsymbol{\sigma}}\right\} + \frac{\partial Q}{\partial J_2}\left\{\frac{\partial J_2}{\partial \boldsymbol{\sigma}}\right\} + \frac{\partial Q}{\partial J_3}\left\{\frac{\partial J_3}{\partial \boldsymbol{\sigma}}\right\} \tag{6.24}$$

where $J_2 = t^2/2$ and the viscoplastic strain rate given by equation (6.19) is evaluated numerically by an expression of the form,

$$\left\{\dot{\boldsymbol{\epsilon}}^{vp}\right\} = F\left(\frac{\partial Q}{\partial \sigma_m}[\mathbf{M}^1] + \frac{\partial Q}{\partial J_2}[\mathbf{M}^2] + \frac{\partial Q}{\partial J_3}[\mathbf{M}^3]\right)\{\boldsymbol{\sigma}\} \tag{6.25}$$

where $\partial Q/\partial \sigma_m$, $\partial Q/\partial J_2$, and $\partial Q/\partial J_3$ are represented by variables `dq1`, `dq2`, and `dq3` in the computer programs, and $\{\partial \sigma_m/\partial \boldsymbol{\sigma}\}$, $\{\partial J_2/\partial \boldsymbol{\sigma}\}$, and $\{\partial J_3/\partial \boldsymbol{\sigma}\}$ by the matrix–vector products $[\mathbf{M}^1]\{\boldsymbol{\sigma}\}$, $[\mathbf{M}^2]\{\boldsymbol{\sigma}\}$, and $[\mathbf{M}^3]\{\boldsymbol{\sigma}\}$. This is essentially the notation used by Zienkiewicz and Taylor (1989), and these quantities are given in more detail in Appendix C.

The body loads $\{\mathbf{F}_b\}^i$ are accumulated at each “time step” within each load step by summing the following integrals for all elements containing a yielding ($F > 0$) Gauss point:

$$\{\mathbf{F}_b\}^i = \{\mathbf{F}_b\}^{i-1} + \sum_{elements}^{all} \iint [\mathbf{B}]^{\mathrm{T}}[\mathbf{D}^e]\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}^i \,\mathrm{d}x\,\mathrm{d}y \tag{6.26}$$

This process is repeated at each “time step” iteration until no integrating point stresses violate the failure criterion within a certain tolerance. The convergence criterion is based on a dimensionless measure of the amount by which the displacement increment vector $\{\mathbf{U}\}^i$ changes from one iteration to the next. The convergence checking process is identical to that used in Program 4.5.

6.7 Initial stress

The viscoplastic algorithm is often referred to as an *initial strain* method to distinguish it from the more widely used "initial stress" approaches (e.g. Zienkiewicz *et al.*, 1969).

Initial stress methods involve an explicit relationship between increments of stress and increments of strain. Thus, whereas linear elasticity was described by

$$\{\mathbf{\Delta}\boldsymbol{\sigma}\} = [\mathbf{D}^e]\left\{\mathbf{\Delta}\boldsymbol{\epsilon}^e\right\} \tag{6.27}$$

elasto-plasticity is described by

$$\{\mathbf{\Delta}\boldsymbol{\sigma}\} = [\mathbf{D}^{pl}]\left\{\mathbf{\Delta}\boldsymbol{\epsilon}^e\right\} \tag{6.28}$$

where

$$[\mathbf{D}^{pl}] = [\mathbf{D}^e] - [\mathbf{D}^p] \tag{6.29}$$

For perfect plasticity in the absence of hardening or softening it is assumed that once a stress state reaches a failure surface, subsequent changes in stress may shift the stress state to a different position on the failure surface, but not outside it, thus

$$\left\{\frac{\partial F}{\partial \boldsymbol{\sigma}}\right\}^{\mathrm{T}}\{\mathbf{\Delta}\boldsymbol{\sigma}\} = 0 \tag{6.30}$$

Allowing for the possibility of non-associated flow, plastic strain increments occur normal to a plastic potential surface, thus

$$\left\{\mathbf{\Delta}\boldsymbol{\epsilon}^p\right\} = \lambda\left\{\frac{\partial Q}{\partial \boldsymbol{\sigma}}\right\} \tag{6.31}$$

Assuming stress changes are generated by elastic strain components only gives

$$\{\mathbf{\Delta}\boldsymbol{\sigma}\} = [\mathbf{D}^e]\left(\{\mathbf{\Delta}\boldsymbol{\epsilon}\} - \lambda\left\{\frac{\partial Q}{\partial \boldsymbol{\sigma}}\right\}\right) \tag{6.32}$$

Substitution of equation (6.32) into (6.30) leads to

$$[\mathbf{D}^p] = \frac{[\mathbf{D}^e]\left\{\frac{\partial Q}{\partial \boldsymbol{\sigma}}\right\}\left\{\frac{\partial F}{\partial \boldsymbol{\sigma}}\right\}^{\mathrm{T}}[\mathbf{D}^e]}{\left\{\frac{\partial F}{\partial \boldsymbol{\sigma}}\right\}^{\mathrm{T}}[\mathbf{D}^e]\left\{\frac{\partial Q}{\partial \boldsymbol{\sigma}}\right\}} \tag{6.33}$$

Explicit versions of $[\mathbf{D}^p]$ may be obtained for simple failure and potential functions and these are given for von Mises (Yamada *et al.*, 1968) and Mohr–Coulomb (Griffiths and Willson, 1986). See Appendix C for a detailed derivation of (6.33)

The body loads $\{\mathbf{F}_b\}^i$ in the stress redistribution process are reformed at each iteration by summing the following integral for all elements that possess yielding Gauss points, thus

$$\{\mathbf{F}_b\}^i = \sum_{elements}^{all}\iint[\mathbf{B}]^{\mathrm{T}}[\mathbf{D}^p]\{\mathbf{\Delta}\boldsymbol{\epsilon}\}^i\ \mathrm{d}x\ \mathrm{d}y \tag{6.34}$$

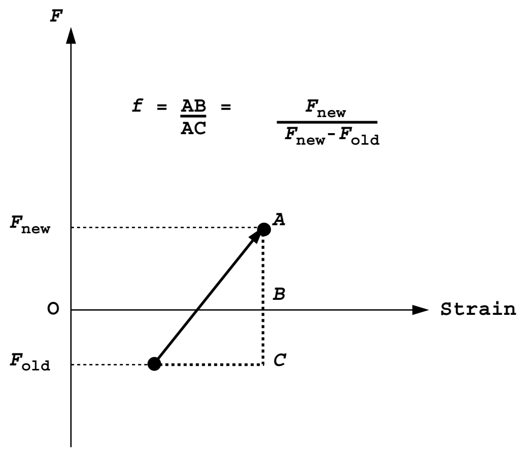


Figure 6.7 Factoring process for "just yielded" elements

In the event of a loading increment causing a Gauss point to go plastic for the first time, it may be necessary to factor the matrix $[\mathbf{D}^p]$ in (6.34). A linear interpolation can be used as indicated in Figure 6.7. Thus, instead of using $[\mathbf{D}^p]$ we use f $[\mathbf{D}^p]$, where

$$f = \frac{F_{\text{new}}}{F_{\text{new}} - F_{\text{old}}} \tag{6.35}$$

This simple method represents a forward Euler approach to integrating the elasto-plastic rate equations, extrapolating from the point at which the yield surface is crossed. More complicated integrations, which are mainly relevant to tangent stiffness methods, are given later in Section 6.9.

Although overshoot of the yield function F is an integral part of the viscoplastic algorithm, a similar interpolation method to that described by (6.35) can be used if required to compute the plastic potential derivatives in equation (6.19) using stresses corresponding to $F \approx 0$.

6.8 Corners on the failure and potential surfaces

For failure and potential surfaces that include "corners" as in Mohr–Coulomb (see Figure 6.6) the derivatives required in equations (6.19) and (6.33) become indeterminate. In the case of the Mohr–Coulomb (or Tresca) surface, this occurs when the angular invariant $\theta = \pm 30^\circ$. The method used in the programs to overcome this difficulty is to replace the hexagonal surface by a smooth conical surface if

$$|\sin\theta| > 0.49 \tag{6.36}$$

The conical surfaces are those obtained by substituting either $\theta = 30^\circ$ or $\theta = -30^\circ$ into (6.12), depending upon the sign of θ as it approaches $\pm 30^\circ$ (see Appendix C). It should be noted that in the initial stress approach, both the F and Q functions must be approximated in this way due to the inclusion of both $\{\partial Q/\partial \boldsymbol{\sigma}\}$ *and* $\{\partial F/\partial \boldsymbol{\sigma}\}$ terms in (6.33). In the viscoplastic algorithm however, only the potential function derivatives $\{\partial Q/\partial \boldsymbol{\sigma}\}$ are approximated since $\{\partial F/\partial \boldsymbol{\sigma}\}$ are not needed by equation (6.19).

All the programs in this chapter for solving two-dimensional problems have been based on the 8-node quadrilateral element, together with reduced integration (four Gauss points per element). This particular combination has been chosen for its simplicity, and also its well-known ability to compute collapse loads accurately (e.g. Zienkiewicz *et al.*, 1975; Griffiths, 1980, 1982). Of course, other element types could be used if required, by making similar changes to those described in Chapter 5. To alleviate "locking" problems with lower order elements, "reduced integration" could be used selectively on the volumetric components of the stiffness matrix (see e.g. Hughes, 1987; Griffiths and Mustoe, 1995).

Program 6.1 Plane strain bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Viscoplastic strain method.

```
PROGRAM p61
!-------------------------------------------------------------------------
! Program 6.1 Plane strain bearing capacity analysis of an elastic-plastic
!             (von Mises) material using 8-node rectangular
!             quadrilaterals. Viscoplastic strain method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,incs,iters,iy,k,limit,loaded_nodes,ndim=2,ndof=16,nels,   &
   neq,nip=4,nn,nod=8,nodof=2,nprops=3,np_types,nr,nst=4,nxe,nye
 REAL(iwp)::ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,d3=3.0_iwp,d4=4.0_iwp,&
   f,lode_theta,one=1.0_iwp,ptot,sigm,tol,two=2.0_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!----------------------dynamic arrays-------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   node(:),num(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   der(:,:),deriv(:,:),devp(:),eld(:),eload(:),eps(:),erate(:),evp(:),    &
   evpt(:,:,:),flow(:,:),g_coord(:,:),jac(:,:),km(:,:),kv(:),loads(:),    &
   m1(:,:),m2(:,:),m3(:,:),oldis(:),points(:,:),prop(:,:),qinc(:),        &
   sigma(:),stress(:),tensor(:,:,:),totd(:),val(:,:),weights(:),          &
   x_coords(:),y_coords(:)
!----------------------input and initialisation---------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),g_g(ndof,nels),  &
   prop(nprops,np_types),etype(nels),evpt(nst,nip,nels),stress(nst),      &
   tensor(nst,nip,nels),coord(nod,ndim),jac(ndim,ndim),der(ndim,nod),     &
   deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof), &
   eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst),       &
   devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq))
!----------------------loop the elements to find global arrays sizes-----
 kdiag=0
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,i7))')                                                   &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); kv=zero
!----------------------element stiffness integration and assembly--------
 elements_2: DO iel=1,nels
   ddt=d4*(one+prop(3,etype(iel)))/(d3*prop(2,etype(iel)))
   IF(ddt<dt)dt=ddt; CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!----------------------read load weightings and factorise equations------
 READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(node(i),val(i,:),i=1,loaded_nodes); CALL sparin(kv,kdiag)
!----------------------load increment loop-------------------------------
 READ(10,*)tol,limit,incs; ALLOCATE(qinc(incs)); READ(10,*)qinc
 WRITE(11,'(/A)')"  step   load       disp      iters"
 oldis=zero; totd=zero; tensor=zero; ptot=zero
 load_incs: DO iy=1,incs
   ptot=ptot+qinc(iy); iters=0; bdylds=zero; evpt=zero
!----------------------plastic iteration loop----------------------------
   its: DO
     iters=iters+1; loads=zero
     WRITE(*,'(A,F8.2,A,I4)')"  load",ptot,"  iteration",iters
     DO i=1,loaded_nodes; loads(nf(:,node(i)))=val(i,:)*qinc(iy); END DO
     loads=loads+bdylds; CALL spabac(kv,loads,kdiag)
!----------------------check plastic convergence-------------------------
     CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
     IF(converged.OR.iters==limit)bdylds=zero
!----------------------go round the Gauss Points ------------------------
     elements_3: DO iel=1,nels
       CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
       num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
       g=g_g(:,iel); eld=loads(g); bload=zero
       gauss_pts_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
         eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
         sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
         CALL invar(stress,sigm,dsbar,lode_theta)
!----------------------check whether yield is violated-------------------
         f=dsbar-SQRT(d3)*prop(1,etype(iel))
         IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
           IF(f>=zero)THEN
             dq1=zero; dq2=d3/two/dsbar; dq3=zero
             CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
             erate=MATMUL(flow,stress); evp=erate*dt
             evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
           END IF
         END IF
         IF(f>=zero.OR.(converged.OR.iters==limit))THEN
           eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
         END IF
!-----------------------update the Gauss Point stresses-------------------
         IF(converged.OR.iters==limit)tensor(:,i,iel)=stress
       END DO gauss_pts_2
!-----------------------compute the total bodyloads vector----------------
       bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
     END DO elements_3; IF(converged.OR.iters==limit)EXIT
   END DO its; totd=totd+loads
   WRITE(11,'(I5,2E12.4,I5)')iy,ptot,totd(nf(2,node(1))),iters
   IF(iters==limit)EXIT
 END DO load_incs
 CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p61
```

Scalar integers:

`i`	simple counter
`iel`	simple counter
`incs`	number of load increments
`iters`	counts plastic iterations
`iwp`	`SELECTED_REAL_KIND(15)`
`iy`	counts load increments
`k`	node number
`limit`	plastic iteration ceiling
`loaded_nodes`	number of loaded nodes
`ndim`	number of dimensions
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integrating points per element
`nn`	number of nodes in the mesh
`nod`	number of nodes per element
`nodof`	number of degrees of freedom per node
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes
`nst`	number of stress (strain) terms
`nxe`	number of elements in x-direction
`nye`	number of elements in y-direction

Scalar reals:

`ddt`	used to find the critical time step
`det`	determinant of the Jacobian matrix
`dq1`	plastic potential derivative, $\partial Q/\partial \sigma_m$
`dq2`	plastic potential derivative, $\partial Q/\partial J_2$

`dq3`	plastic potential derivative, $\partial Q/\partial J_3$
`dsbar`	invariant, $\overline{\sigma}$
`dt`	critical viscoplastic time step (set initially to 10^{15})
`d3`	set to 3.0
`d4`	set to 4.0
`f`	value of yield function
`load_theta`	Lode angle, θ
`one`	set to 1.0
`ptot`	holds running total of applied pressure
`sigm`	mean stress, σ_m
`tol`	plastic convergence tolerance
`two`	set to 2.0
`zero`	set to 0.0

Scalar character:

`element`	element type

Scalar logical:

`converged`	set to `.TRUE.` if plastic iterations have converged

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`kdiag`	diagonal term location vector
`nf`	nodal freedom matrix
`node`	loaded nodes vector
`num`	element node numbers vector

Dynamic real arrays:

`bdylds`	self-equilibrating global body loads
`bee`	strain-displacement matrix
`bload`	self-equilibrating element body loads
`coord`	element nodal coordinates
`dee`	stress–strain matrix
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`devp`	product $[\mathbf{D}^e]\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}$
`eld`	element nodal displacements
`eload`	integrating point contribution to `bload`
`eps`	strain terms
`erate`	viscoplastic strain rate, $\left\{\dot{\boldsymbol{\epsilon}}^{vp}\right\}$
`evp`	viscoplastic strain rate increment, $\left\{\boldsymbol{\delta\epsilon}^{vp}\right\}$
`evpt`	holds running total of viscoplastic strains, $\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}$
`flow`	holds $\{\partial Q/\partial\boldsymbol{\sigma}\}$

`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`km`	element stiffness matrix
`kv`	global stiffness matrix
`loads`	nodal loads and displacements
`m1`	used to compute $\{\partial\sigma_m/\partial\boldsymbol{\sigma}\}$
`m2`	used to compute $\{\partial J_2/\partial\boldsymbol{\sigma}\}$
`m3`	used to compute $\{\partial J_3/\partial\boldsymbol{\sigma}\}$
`oldis`	nodal displacements from previous iteration
`points`	integrating point local coordinates
`prop`	element properties
`qinc`	holds applied pressure increments
`sigma`	stress terms
`stress`	stress term increments
`tensor`	holds running total of all integrating point stress terms
`totd`	holds running total of nodal displacement
`val`	applied nodal load weightings
`weights`	weighting coefficients
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout

Program 6.1 employs the viscoplastic method to compute the response to loading of an elastic-perfectly plastic von Mises (6.7) material. Plane strain conditions are enforced and, in order to monitor the load-displacement response, the loads are applied incrementally. As in Program 4.5, the method uses constant stiffness iterations, thus the relatively time-consuming subroutine `sparin` is called just once, while the subroutine `spabac` is called at each iteration. An outline of the viscoplastic algorithm which comes after the stiffness matrix formation is given in the structure chart in Figure 6.8.

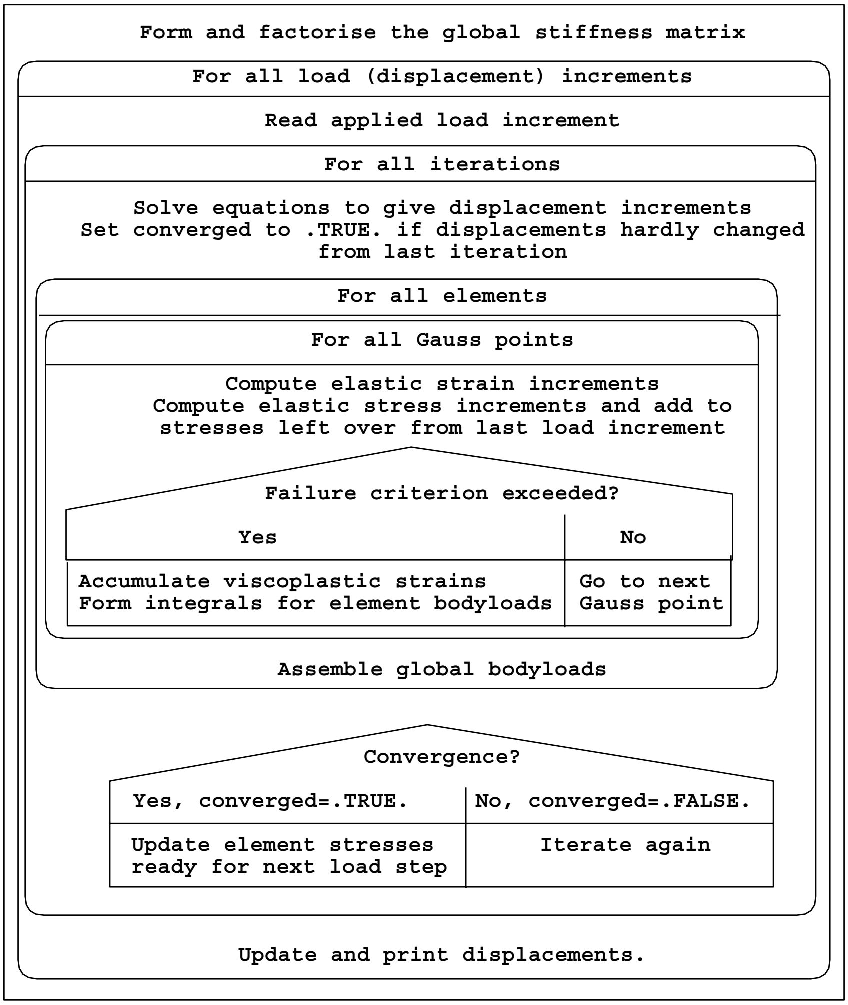
This program uses 8-node quadrilateral elements with numbering in the y-direction. The familiar subroutine `geom_rect`, produces the mesh of rectangular 8-node elements, with the numbering direction, in this case, entered explicitly in the argument list as `'y'`. Subroutines not seen previously include `invar`, which forms the three invariants (σ_m, $\overline{\sigma}$, θ) (6.3) to (6.4) from the four Cartesian stress components held in `stress`. It should be noted that in plane strain plasticity applications, it is necessary to retain four components of stress and strain. Although, by definition, ϵ_z must equal zero, the elastic strain in that direction may be non-zero provided,

$$\epsilon_z^e = -\epsilon_z^{vp} \tag{6.37}$$

is satisfied. For this reason, the 4×4 (2.77) elastic stress–strain matrix $[\mathbf{D}^e]$ is provided by the subroutine `deemat` with `nst = 4`.

The only other subroutine not encountered before is `formm`, which creates arrays `m1`, `m2`, and `m3` used in the calculation of the viscoplastic strain rate from equation (6.25).

The example shown in Figure 6.9 is of a flexible strip footing at the surface of a layer of uniform undrained clay. The footing supports a uniform stress, q, which is increased incrementally to failure. The elasto-plastic soil is described by three parameters (`nprops=3`), namely the undrained "cohesion" c_u, followed by the elastic properties,



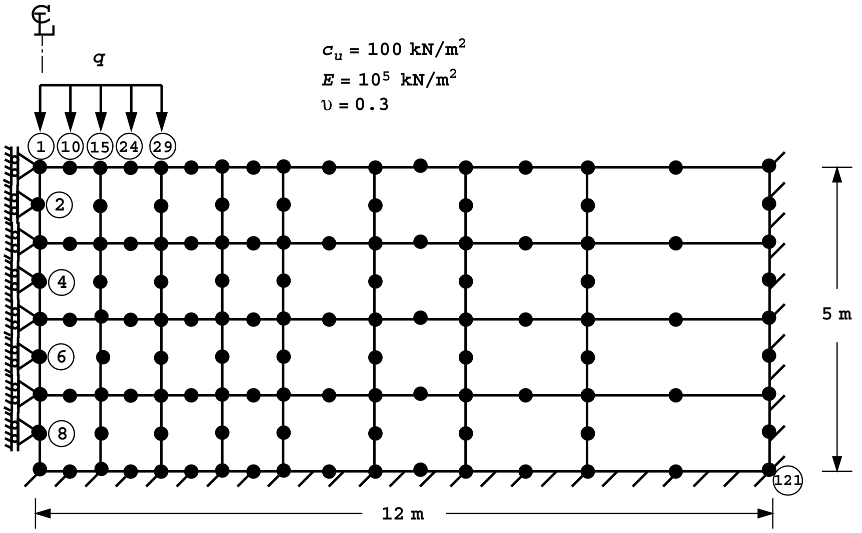
```
Form and factorise the global stiffness matrix
For all load (displacement) increments
    Read applied load increment
    For all iterations
        Solve equations to give displacement increments
        Set converged to .TRUE. if displacements hardly changed
        from last iteration
        For all elements
            For all Gauss points
                Compute elastic strain increments
                Compute elastic stress increments and add to
                stresses left over from last load increment
                Failure criterion exceeded?
                    Yes
                        Accumulate viscoplastic strains
                        Form integrals for element bodyloads
                    No
                        Go to next
                        Gauss point
            Assemble global bodyloads
        Convergence?
            Yes, converged=.TRUE.
                Update element stresses
                ready for next load step
            No, converged=.FALSE.
                Iterate again
    Update and print displacements.
```

Figure 6.8 Structure chart for viscoplastic algorithm

E and v. Theoretically, bearing failure in this problem occurs when q reaches the “Prandtl” load given by

$$q_{ult} = (2 + \pi)c_u \tag{6.38}$$

Apart from the variables, `type_2d='plane'`, `element='quadrilateral'`, `nod=8` and `dir='y'`, which are built into the program, the data follows the familiar pattern established in Chapter 5. The “loads” in this case are the nodal forces which would deliver a uniform stress of 1 kN/m^2 across the footing semi-width of 2 m (Appendix A). These “weightings” are then increased proportionally by the load increment values held in the vector `qinc`. In order to capture failure in a load-controlled problem such as this, the increments need to made smaller as failure is approached. This may involve some trial



```
nxe   nye   np_types
8     4     1

prop(cu,e,v)
100.0  1.0e5  0.3

etype(not needed)

x_coords, y_coords
0.0   1.0   2.0   3.0   4.0   5.5   7.0   9.0  12.0
0.0  -1.25 -2.5  -3.75 -5.0

nr,(k,nf(:,k),i=1,nr)
33
  1 0 1    2 0 1    3 0 1    4 0 1    5 0 1    6 0 1
  7 0 1    8 0 1    9 0 0   14 0 0   23 0 0   28 0 0
 37 0 0   42 0 0   51 0 0   56 0 0   65 0 0   70 0 0
 79 0 0   84 0 0   93 0 0   98 0 0  107 0 0  112 0 0
113 0 0  114 0 0  115 0 0  116 0 0  117 0 0  118 0 0
119 0 0  120 0 0  121 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
5
1   0.0  -0.166667     10  0.0  -0.666667     15  0.0  -0.333333
24  0.0  -0.666667     29  0.0  -0.166667

tol   limit
0.001  250

incs,(qinc(i),i=1,incs)
10
200.0 100.0  50.0  50.0  50.0  30.0  20.0  10.0   5.0   5.0
```

Figure 6.9 Mesh and data for Program 6.1 example

and error on the part of the user in an unfamiliar problem. New input variables involve `tol`, the convergence tolerance (set to 0.001), and `limit`, the iteration ceiling, set to 250, representing the maximum number of iterations (`iters`) that will be allowed within any load increment. If `iters` ever becomes equal to `limit`, the algorithm stops and no more load increments are applied.

```
There are    184 equations and the skyline storage is   4130

 step   load        disp      iters
   1  0.2000E+03 -0.6592E-02    2
   2  0.3000E+03 -0.1155E-01   11
   3  0.3500E+03 -0.1630E-01   20
   4  0.4000E+03 -0.2316E-01   33
   5  0.4500E+03 -0.3317E-01   45
   6  0.4800E+03 -0.4227E-01   65
   7  0.5000E+03 -0.5084E-01   81
   8  0.5100E+03 -0.5665E-01   99
   9  0.5150E+03 -0.6093E-01  159
  10  0.5200E+03 -0.7459E-01  250
```

Figure 6.10 Results from Program 6.1 example

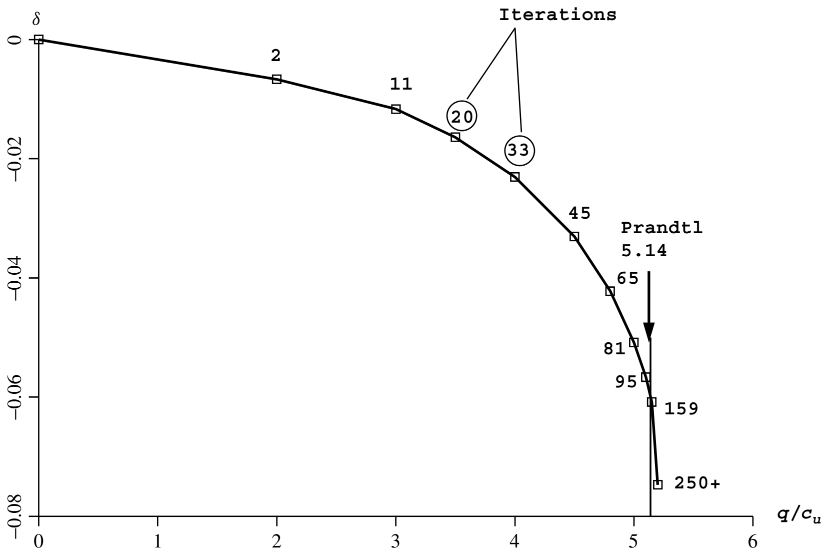


Figure 6.11 Plot of bearing stress versus centreline displacement

At load levels well below failure, convergence should occur in relatively few iterations. As failure is approached, the algorithm has to work harder and requires more iterations to converge. The computed results for this example are given in Figure 6.10, and show the applied stress, the vertical displacement under the centreline and the number of iterations for convergence. These results have been plotted in Figure 6.11 in the form of a dimensionless bearing capacity factor q/c_u versus centreline displacement. The number of iterations to achieve convergence for each load increment is also shown. It is seen that convergence was achieved in 159 iterations when $q/c_u = 5.1$, but convergence could not be achieved within the upper limit of 250 when $q/c_u = 5.2$. In addition, the displacements are also increasing rapidly at this level of loading, indicating that bearing failure is taking place at a value very close to the "Prandtl" load of 5.14.

Program 6.1 creates the graphical output files `fe95.msh` (undeformed mesh), `fe95.dis` (deformed mesh), and `fe95.vec` (nodal displacement vectors) first encountered in Chapter 5. Although the relatively crude mesh of Figures 6.9 gave an

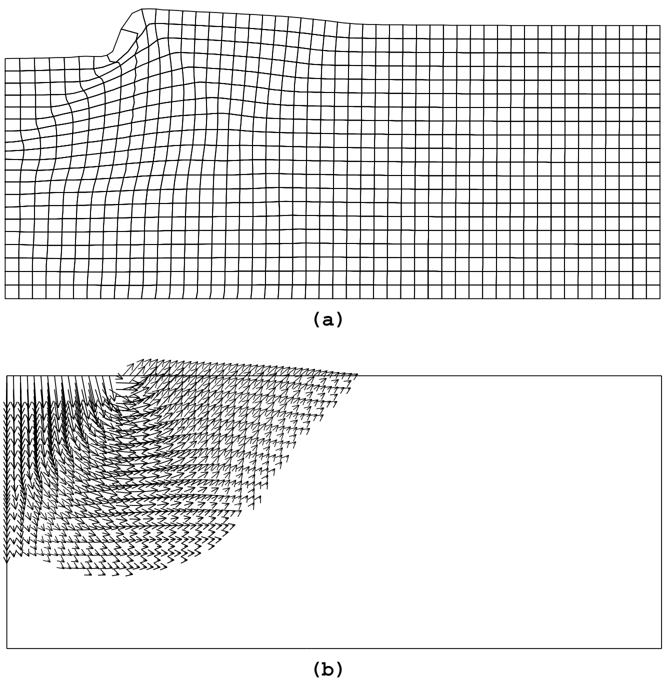


Figure 6.12 Graphical output from Program 6.1. (a) Deformed mesh and (b) nodal displacement vectors at bearing failure

accurate estimate of the failure loading, Figures 6.12(a) and (b) show the deformed mesh and displacement vectors corresponding to the unconverged "solution" at failure using a rather more refined mesh. The displacements are uniformly magnified to emphasise the deformations. Although the finite element mesh is constrained to remain a continuum, the figures are still able to give a good indication of the form of the failure mechanism.

Program 6.2 Plane strain bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Viscoplastic strain method. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver.

```
PROGRAM p62
!-------------------------------------------------------------------------
! Program 6.2 Plane strain bearing capacity analysis of an elastic-plastic
!             (von Mises) material using 8-node rectangular
!             quadrilaterals. Viscoplastic strain method. No global
!             stiffness matrix assembly. Diagonally preconditioned
!             conjugate gradient solver.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,cg_tot,i,iel,incs,iters,iy,k,limit,         &
   loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nn,nod=8,nodof=2,nprops=3,  &
   np_types,nr,nst=4,nxe,nye
 REAL(iwp)::alpha,beta,cg_tol,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,    &
  d3=3.0_iwp,d4=4.0_iwp,f,lode_theta,one=1.0_iwp,ptot,sigm,tol,           &
  two=2.0_iwp,up,zero=0.0_iwp; CHARACTER(LEN=15)::element='quadrilateral'
 LOGICAL::converged,cg_converged
!-----------------------dynamic arrays----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),nf(:,:),node(:),  &
   num(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),d(:),      &
   dee(:,:),der(:,:),deriv(:,:),devp(:),diag_precon(:),eld(:),eload(:),   &
   eps(:),erate(:),evp(:),evpt(:,:,:),flow(:,:),g_coord(:,:),jac(:,:),    &
   km(:,:),kv(:),loads(:),m1(:,:),m2(:,:),m3(:,:),oldis(:),p(:),          &
   points(:,:),prop(:,:),qinc(:),sigma(:),storkm(:,:,:),stress(:),        &
   tensor(:,:,:),totd(:),u(:),val(:,:),weights(:),x(:),xnew(:),           &
   x_coords(:),y_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,cg_tol,cg_limit,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),coord(nod,ndim), &
   evpt(nst,nip,nels),tensor(nst,nip,nels),etype(nels),jac(ndim,ndim),    &
   der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),           &
   km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof),eload(ndof),   &
   erate(nst),evp(nst),devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),         &
   m3(nst,nst),flow(nst,nst),stress(nst),g_g(ndof,nels),                  &
   storkm(ndof,ndof,nels),prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 WRITE(11,'(A,I7,A)')"There are",neq," equations"
 ALLOCATE(loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq),p(0:neq),   &
   x(0:neq),xnew(0:neq),u(0:neq),diag_precon(0:neq),d(0:neq))
!-----------------------loop the elements to set up element data----------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 CALL sample(element,points,weights); diag_precon=zero
!----------element stiffness integration, storage and preconditioner------
 elements_2: DO iel=1,nels
   ddt=d4*(one+prop(3,etype(iel)))/(d3*prop(2,etype(iel)))
   if(ddt<dt)dt=ddt; CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1; storkm(:,:,iel)=km
   DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
 END DO elements_2; diag_precon(1:)=one/diag_precon(1:)
!-----------------------read load weightings-----------------------------
 READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
!-----------------------load increment loop------------------------------
 READ(10,*)tol,limit,incs; ALLOCATE(qinc(incs)); READ(10,*)qinc
 WRITE(11,'(/A)')                                                         &
```

```
   "  step   load        disp      iters    cg iters/plastic iter"
 oldis=zero; totd=zero; tensor=zero; ptot=zero; diag_precon(0)=zero
 load_incs: DO iy=1,incs
   ptot=ptot+qinc(iy); iters=0; bdylds=zero; evpt=zero; cg_tot=0
!-----------------------plastic iteration loop----------------------------
   its: DO
     iters=iters+1; loads=zero
     WRITE(*,'(A,F8.2,A,I4)')"  load",ptot,"  iteration",iters
     DO i=1,loaded_nodes; loads(nf(:,node(i)))=val(i,:)*qinc(iy); END DO
     loads=loads+bdylds; d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----------------------pcg equation solution-----------------------------
     pcg: DO
       cg_iters=cg_iters+1; u=zero
       elements_3 : DO iel=1,nels
         g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
       END DO elements_3
       up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
       loads=loads-u*alpha; d=diag_precon*loads
       beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
       call checon(xnew,x,cg_tol,cg_converged)
       IF(cg_converged.OR.cg_iters==cg_limit)EXIT
     END DO pcg; cg_tot=cg_tot+cg_iters; loads=xnew; loads(0)=zero
!-----------------------check plastic convergence-------------------------
     CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
     IF(converged.OR.iters==limit)bdylds=zero
!-----------------------go round the Gauss Points-------------------------
     elements_4: DO iel=1,nels
       CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
       num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
       g=g_g(:,iel); eld=loads(g); bload=zero
       gauss_pts_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
         eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
         sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
         CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         f=dsbar-SQRT(d3)*prop(1,etype(iel))
         IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
           IF(f>=zero)THEN
             dq1=zero; dq2=d3/two/dsbar; dq3=zero
             CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
             erate=MATMUL(flow,stress); evp=erate*dt
             evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
           END IF
         END IF
         IF(f>=zero)THEN
           eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
         END IF
!-----------------------update the Gauss Point stresses-------------------
         IF(converged.OR.iters==limit)tensor(:,i,iel)=stress
       END DO gauss_pts_2
!-----------------------compute the total bodyloads vector----------------
       bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
     END DO elements_4; IF(converged.OR.iters==limit)EXIT
   END DO its; totd=totd+loads
   WRITE(11,'(I5,2E12.4,I5,F17.2)')                                     &
```

```
      iy,ptot,totd(nf(2,node(1))),iters,REAL(cg_tot)/REAL(iters)
    IF(iters==limit)THEN
      CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
      CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
      STOP
    END IF
  END DO load_incs
STOP
END PROGRAM p62
```

New scalar integers:

`cg_iters`	conjugate gradient iteration counter
`cg_limit`	conjugate gradient iteration ceiling
`cg_tot`	keeps running total of `cg_iters`

New scalar reals:

`alpha`	α from equations (3.22)
`beta`	β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\{\mathbf{R}\}_k$ from equations (3.22)

New Scalar logical:

`cg_converged`	set to `.TRUE.` if pcg process has converged

New dynamic real arrays:

`d`	preconditioned rhs vector
`diag_precon`	diagonal preconditioner vector
`p`	"descent" vector used in equations (3.22)
`storkm`	holds element stiffness matrices
`u`	vector used in equations (3.22)
`x`	"old" solution vector
`xnew`	"new" solution vector

For non-linear problems, computation times are now more demanding. For example Program 6.1 required some 765 linear elastic solutions to reach the failure load. Clearly vectorised and parallelised algorithms will become essential for much larger problems. The mesh-free approach first demonstrated in Program 5.5 will be attractive for large non-linear problems and is implemented in Program 6.2. The data in Figure 6.13 are identical to those used in Program 6.1 with the addition of a conjugate gradient convergence tolerance `cg_tol` set to 0.0001, and conjugate gradient iteration ceiling `cg_limit` set to 100. The results are shown in Figure 6.14, and are nearly identical to those in Figure 6.10. The results table in Figure 6.14 also indicates that approximately 50 conjugate gradient iterations were needed on average for each plastic iteration. In scalar computations, therefore, this algorithm will not be attractive, but it does have some strong attractions for parallel computation. Because a "constant stiffness" approach is being used, groups of elements in Figure 6.9 have

```
nxe   nye   cg_tol   cg_limit   np_types
8     4     0.0001    100       1

prop(cu,e,v)
100.0  1.0e5  0.3

etype(not needed)

x_coords, y_coords
0.0   1.0   2.0   3.0   4.0   5.5   7.0   9.0  12.0
0.0  -1.25 -2.5  -3.75 -5.0

nr,(k,nf(:,k),i=1,nr)
33
  1 0 1    2 0 1    3 0 1    4 0 1    5 0 1    6 0 1
  7 0 1    8 0 1    9 0 0   14 0 0   23 0 0   28 0 0
 37 0 0   42 0 0   51 0 0   56 0 0   65 0 0   70 0 0
 79 0 0   84 0 0   93 0 0   98 0 0  107 0 0  112 0 0
113 0 0  114 0 0  115 0 0  116 0 0  117 0 0  118 0 0
119 0 0  120 0 0  121 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
5
1   0.0  -0.166667     10  0.0  -0.666667     15  0.0  -0.333333
24  0.0  -0.666667     29  0.0  -0.166667

tol   limit
0.001  250

incs,(qinc(i),i=1,incs)
10
200.0 100.0  50.0  50.0  50.0  30.0  20.0  10.0   5.0   5.0
```

Figure 6.13 Data for Program 6.2 example

```
 There are    184 equations

   step   load        disp      iters    cg iters/plastic iter
     1  0.2000E+03 -0.6593E-02    2            46.00
     2  0.3000E+03 -0.1155E-01   11            46.82
     3  0.3500E+03 -0.1629E-01   20            50.70
     4  0.4000E+03 -0.2316E-01   33            51.21
     5  0.4500E+03 -0.3317E-01   45            52.60
     6  0.4800E+03 -0.4228E-01   65            53.23
     7  0.5000E+03 -0.5086E-01   82            54.05
     8  0.5100E+03 -0.5667E-01   98            53.54
     9  0.5150E+03 -0.6091E-01  147            52.39
    10  0.5200E+03 -0.7473E-01  250            53.77
```

Figure 6.14 Results from Program 6.2 example

constant properties throughout the calculation (there are only 4 distinct element types in this case). This feature can also be exploited in the parallel algorithms described in Chapter 12. Also, computation costs can be approximately halved by using the previously computed `bdylds` rather than setting `bdylds` to zero at the beginning of each load increment.

Program 6.3 Plane strain slope stability analysis of an elastic–plastic (Mohr–Coulomb) material using 8-node rectangular quadrilaterals. Viscoplastic strain method.

```
PROGRAM p63
!-------------------------------------------------------------------------
! Program 6.3 Plane strain slope stability analysis of an elastic-plastic
!             (Mohr-Coulomb) material using 8-node rectangular
!             quadrilaterals. Viscoplastic strain method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,iters,iy,limit,ndim=2,ndof=16,nels,neq,nip=4,nn,nod=8,    &
   nodof=2,nprops=6,np_types,nsrf,nst=4,nx1,nx2,nye,ny1,ny2
 REAL(iwp)::cf,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,d4=4.0_iwp,        &
   d180=180.0_iwp,e,f,fmax,h1,h2,lode_theta,one=1.0_iwp,phi,phif,pi,psi,  &
   psif,sigm,snph,start_dt=1.e15_iwp,s1,tnph,tnps,tol,two=2.0_iwp,v,w1,w2,&
   zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   num(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   devp(:),elastic(:),eld(:),eload(:),eps(:),erate(:),evp(:),evpt(:,:,:), &
   flow(:,:),fun(:),gravlo(:),g_coord(:,:),km(:,:),kv(:),loads(:),m1(:,:),&
   m2(:,:),m3(:,:),oldis(:),points(:,:),prop(:,:),sigma(:),srf(:),        &
   weights(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)w1,s1,w2,h1,h2,nx1,nx2,ny1,ny2,np_types; nye=ny1+ny2
 nels=nx1*nye+ny2*nx2; nn=(3*nye+2)*nx1+2*nye+1+(3*ny2+2)*nx2
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod),     &
   g_g(ndof,nels),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof),  &
   eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst),       &
   devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),   &
   prop(nprops,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 CALL emb_2d_bc(nx1,nx2,ny1,ny2,nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),             &
   gravlo(0:neq),elastic(0:neq)); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL emb_2d_geom(iel,nx1,nx2,ny1,ny2,w1,s1,w2,h1,h2,coord,num)
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,i7))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); kv=zero; gravlo=zero
!-----------------------element stiffness integration and assembly--------
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; eld=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i)
     CALL bee8(bee,coord,points(i,1),points(i,2),det)
```

```
    km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    eld(nodof:ndof:nodof)=eld(nodof:ndof:nodof)+fun(:)*det*weights(i)
  END DO gauss_pts_1
  CALL fsparv(kv,km,g,kdiag); gravlo(g)=gravlo(g)-eld*prop(4,etype(iel))
 END DO elements_2
!-----------------------factorise equations------------------------------
 CALL sparin(kv,kdiag); pi=ACOS(-one)
!-----------------------trial strength reduction factor loop--------------
 READ(10,*)tol,limit,nsrf; ALLOCATE(srf(nsrf)); READ(10,*)srf
 WRITE(11,'(/A)')"    srf    max disp  iters"
 srf_trials: DO iy=1,nsrf
   dt=start_dt
   DO i=1,np_types
     phi=prop(1,i); tnph=TAN(phi*pi/d180); phif=ATAN(tnph/srf(iy))
     snph=SIN(phif); e=prop(5,i); v=prop(6,i)
     ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
   END DO; iters=0; bdylds=zero; evpt=zero; oldis=zero
!-----------------------plastic iteration loop----------------------------
   its: DO
     fmax=zero; iters=iters+1; loads=gravlo+bdylds
     CALL spabac(kv,loads,kdiag); loads(0)=zero
     IF(iy==1.AND.iters==1)elastic=loads
!-----------------------check plastic convergence-------------------------
     CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
     IF(converged.OR.iters==limit)bdylds=zero
!-----------------------go round the Gauss Points ------------------------
     elements_3: DO iel=1,nels
       bload=zero; phi=prop(1,etype(iel)); tnph=TAN(phi*pi/d180)
       phif=ATAN(tnph/srf(iy))*d180/pi; psi=prop(3,etype(iel))
       tnps=TAN(psi*pi/d180); psif=ATAN(tnps/srf(iy))*d180/pi
       cf=prop(2,etype(iel))/srf(iy); e=prop(5,etype(iel))
       v=prop(6,etype(iel)); CALL deemat(dee,e,v); num=g_num(:,iel)
       coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
       gauss_pts_2: DO i=1,nip
         CALL bee8(bee,coord,points(i,1),points(i,2),det)
         eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel); sigma=MATMUL(dee,eps)
         CALL invar(sigma,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         CALL mocouf(phif,cf,sigm,dsbar,lode_theta,f); IF(f>fmax)fmax=f
         IF(converged.OR.iters==limit)THEN; devp=sigma; ELSE
           IF(f>=zero.OR.(converged.OR.iters==limit))THEN
             CALL mocouq(psif,dsbar,lode_theta,dq1,dq2,dq3)
             CALL formm(sigma,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
             erate=MATMUL(flow,sigma); evp=erate*dt
             evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
           END IF
         END IF
         IF(f>=zero)THEN
           eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
         END IF
       END DO gauss_pts_2
!-----------------------compute the total bodyloads vector----------------
       bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
     END DO elements_3
     WRITE(*,'(A,F7.2,A,I4,A,F8.3)')                                    &
       "  srf",srf(iy),"  iteration",iters,"  F_max",fmax
     IF(converged.OR.iters==limit)EXIT
   END DO its; WRITE(11,'(F7.2,E12.4,I5)')srf(iy),MAXVAL(ABS(loads)),iters
   IF(iters==limit)EXIT
 END DO srf_trials
 CALL dismsh(loads-elastic,nf,0.1_iwp,g_coord,g_num,13)
 CALL vecmsh(loads-elastic,nf,0.1_iwp,0.25_iwp,g_coord,g_num,14)
STOP
END PROGRAM p63
```

New scalar integers:

`nsrf`	number of trial strength reduction factors
`nx1`	number of "columns" of elements in embankment
`nx2`	number of columns of elements to right of toe
`ny1`	number of rows of elements in embankment
`ny2`	number of rows of elements in foundation

New scalar reals:

`cf`	factored cohesion
`d180`	set to 180.0
`e`	Young's modulus
`fmax`	maximum value of `F`
`h1`	height of embankment
`h2`	height of foundation
`phi`	friction angle (degrees)
`phif`	factored friction angle
`pi`	set to π
`psi`	dilation angle (degrees)
`psif`	factored dilation angle
`snph`	sin of `phi`
`start_dt`	starting value of `dt`
`s1`	width of top of embankment
`tnph`	tangent of `phi`
`tnps`	tangent of `psi`
`v`	Poisson's ratio
`w1`	width of sloping section of embankment
`w2`	distance foundation extends beyond the toe

New dynamic real arrays:

`elastic`	elastic nodal displacements
`fun`	shape functions
`gravlo`	loads generated by gravity
`srf`	trial strength reduction factors

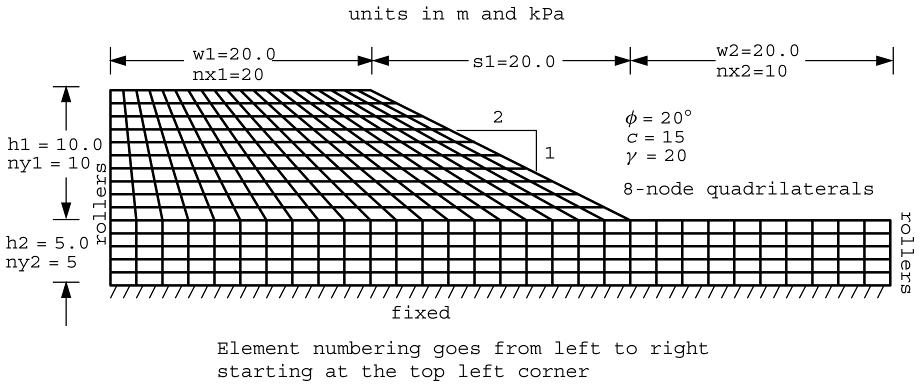
This program is, in many ways, similar to Program 6.1. The problem to be analysed is a slope of Mohr–Coulomb material subjected to gravity loading. The factor of safety (FS) of the slope is to be assessed, and this quantity is defined as the proportion by which $\tan\phi$ and c must be reduced in order to cause failure with the gravity loading held constant. This is in contrast to the previous programs in this chapter in which failure was induced by increasing the loads with the material properties remaining constant.

Gravity loads are generated in the manner described in Chapter 5 (5.7) and applied to the slope in a single increment. A trial strength reduction factor loop gradually weakens the soil until the algorithm fails to converge. Each entry of this loop implements a different strength reduction factor *SRF*. The factored soil strength parameters that go into the elasto-plastic analysis are obtained from,

$$\phi_f = \arctan(\tan\phi/SRF)$$
$$c_f = c/SRF \qquad (6.39)$$

Several (usually increasing) values of the *SRF* factor are attempted until the algorithm fails to converge, at which point SRF is then interpreted as the factor of safety *FS*. For a detailed description of the algorithm, the reader is referred to Griffiths and Lane (1999)

Subroutines new to this program include `emb_2d_geom` and `emb_2d_bc`. These subroutines generate the mesh and boundary conditions for a standard slope cross-section of the type shown in Figure 6.15, with dimensions and mesh density controlled through the



```
w1    s1    w2    h1    h2
20.0  20.0  20.0  10.0  5.0

nx1  nx2  ny1  ny2
20   10   10    5

np_types
1

prop(phi,c,psi,gamma,e,v)
20.0  15.0  0.0  20.0  1.0e5  0.3

etype(not needed)

tol    limit
0.0001  500

nsrf,(srf(i),i=1,nsrf)
6
1.0  1.2  1.4  1.5  1.55 1.6
```

Figure 6.15 Mesh and data for Program 6.3 example

input data. The boundary conditions are rollers on the left and right vertical boundaries, and full fixity at the base.

Subroutine mocouf computes the Mohr–Coulomb failure function F from the current stress state and the factored shear strength parameters (6.12). Subroutine mocouq forms the derivatives of the Mohr–Coulomb potential function Q with respect to the three stress invariants and these values are held in dq1, dq2, and dq3. In Programs 6.1 and 6.2, similar subroutines corresponding to the von Mises criterion could have been used, but the required expressions were so trivial that they were written directly into the main program.

For each material type, six properties must be read in (nprops=6); the friction angle ϕ, the cohesion c, the dilation angle ψ, the total unit weight γ, Young's modulus E, and Poisson's ratio ν.

Figure 6.15 shows the mesh and data for an analysis of a homogeneous 2:1 slope with $\phi = 20^\circ$ and $c = 15$ kN/m^2. The dilation angle ψ is set to zero and the unit weight is given

```
There are   2120 equations and the skyline storage is 151000

  srf    max disp  iters
 1.00  0.1711E-01   10
 1.20  0.1889E-01   17
 1.40  0.2115E-01   33
 1.50  0.2283E-01   67
 1.55  0.2446E-01  244
 1.60  0.3761E-01  500
```

Figure 6.16 Results from Program 6.3 example

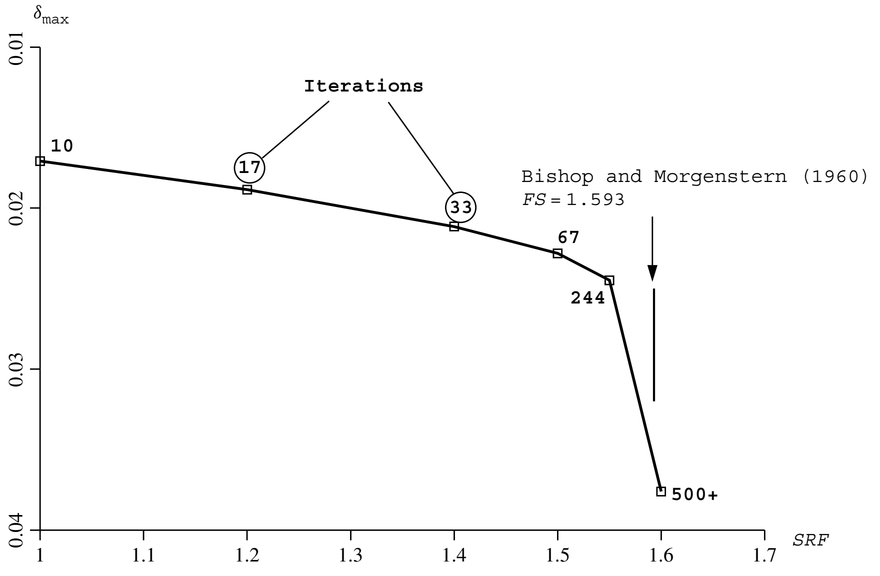


Figure 6.17 Plot of maximum displacement versus Strength Reduction Factor from Program 6.3 example

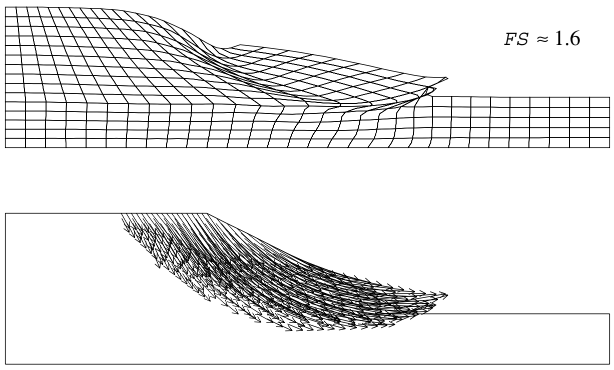


Figure 6.18 Deformed mesh and displacement vectors at failure from Program 6.3 example

as $\gamma = 20$ kN/m^3. The elastic parameters are given nominal values of $E = 1 \times 10^5$ kN/m^2 and $\nu = 0.3$ since they have little influence on the computed factor of safety. The convergence tolerance and iteration ceiling are set to `tol=0.0001` and `limit=500` respectively. Six trial strength reduction factors (`nsrf=6`) are input, ranging from 1.0 to 1.6.

No `etype` data is required in this homogeneous example, but if it is required, the user needs to know that element numbering proceeds in the x-direction, starting at the top-left corner of the mesh.

The output in Figure 6.16 gives the strength reduction factor, the maximum nodal displacement at convergence, and the number of iterations to achieve convergence. It can be seen that when `srf=1.6`, the iteration ceiling of 500 was reached. A plot of these results in Figure 6.17 shows that the displacements increase rapidly at this level of strength reduction, indicating a factor of safety of about 1.6. Bishop and Morgenstern's charts (1960) give a factor of safety of 1.593 for the slope under consideration. Figure 6.18 displays the PostScript files `fe95.dis` and `fe95.vec`, which show the deformed mesh and displacement vectors corresponding to slope failure. The mechanism of failure is clearly shown to be of the "toe" type.

Program 6.4 Plane strain earth pressure analysis of an elastic–plastic (Mohr–Coulomb) material using 8-node rectangular quadrilaterals. Initial stress method.

```
PROGRAM p64
!-------------------------------------------------------------------------
! Program 6.4 Plane strain earth pressure analysis of an elastic-plastic
!             (Mohr-Coulomb) material using 8-node rectangular
!             quadrilaterals. Initial stress method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,incs,iters,iy,k,limit,ndim=2,ndof=16,nels, &
   neq,nip=4,nn,nod=8,nodof=2,nprops=7,np_types,nr,nst=4,nxe,nye
 REAL(iwp)::c,det,dsbar,e,f,fac,fnew,gamma,k0,lode_theta,one=1.0_iwp,ot,  &
   pav,phi,psi,pr,presc,pt5=0.5_iwp,sigm,penalty=1.0e20_iwp,tol,v,        &
   zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   der(:,:),deriv(:,:),eld(:),eload(:),elso(:),eps(:),fun(:),gc(:),       &
   g_coord(:,:),jac(:,:),km(:,:),kv(:),loads(:),oldis(:),pl(:,:),         &
   points(:,:),prop(:,:),react(:),rload(:),sigma(:),storkv(:),stress(:),  &
   tensor(:,:,:),totd(:),weights(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),fun(nod),        &
   tensor(nst,nip,nels),g_g(ndof,nels),coord(nod,ndim),stress(nst),       &
   jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),          &
   bee(nst,ndof),km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof), &
   eload(ndof),pl(nst,nst),elso(nst),g(ndof),gc(ndim),rload(ndof),        &
   prop(nprops,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq), &
   react(0:neq)); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I7))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); tensor=zero; kv=zero
!-----------------------element stiffness integration and assembly--------
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(6,etype(iel)),prop(7,etype(iel)))
   gamma=prop(4,etype(iel)); k0=prop(5,etype(iel))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); gc=MATMUL(fun,coord)
     tensor(2,i,iel)=(gc(2)-y_coords(1))*gamma
     tensor(1,i,iel)=(gc(2)-y_coords(1))*gamma*k0
     tensor(4,i,iel)=tensor(1,i,iel); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!-----------------------read displacement data and factorise equations----
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),no(fixed_freedoms),&
     storkv(fixed_freedoms))
   READ(10,*)(node(i),sense(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; storkv=kv(kdiag(no))
 END IF; CALL sparin(kv,kdiag)
```

```
!-----------------------displacement increment loop-----------------------
 READ(10,*)tol,limit,incs,presc
 WRITE(11,'(/A)')                                                       &
   "  step   disp      load(av)    load(react)   moment    iters"
 oldis=zero; totd=zero; bdylds=zero
 disp_incs: DO iy=1,incs
   iters=0; react=zero
!-----------------------plastic iteration loop----------------------------
   its: DO
     iters=iters+1; loads=bdylds
     WRITE(*,'(A,E11.3,A,I4)')"  disp",iy*presc,"  iteration",iters
     DO i=1,fixed_freedoms; loads(nf(1,node(i)))=storkv(i)*presc; END DO
     CALL spabac(kv,loads,kdiag); bdylds=zero
!-----------------------check plastic convergence-------------------------
     CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
!-----------------------go round the Gauss Points ------------------------
     elements_3: DO iel=1,nels
       phi=prop(1,etype(iel)); c=prop(2,etype(iel)); psi=prop(3,etype(iel))
       e=prop(6,etype(iel)); v=prop(7,etype(iel)); CALL deemat(dee,e,v)
       bload=zero; rload=zero; num=g_num(:,iel)
       coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
       gauss_pts_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
         sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
         CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         CALL mocouf(phi,c,sigm,dsbar,lode_theta,fnew); elso=zero
         IF(fnew>zero)THEN
           stress=tensor(:,i,iel); CALL invar(stress,sigm,dsbar,lode_theta)
           CALL mocouf(phi,c,sigm,dsbar,lode_theta,f); fac=fnew/(fnew-f)
           stress=(one-fac)*sigma+tensor(:,i,iel)
           CALL mcdpl(phi,psi,dee,stress,pl); pl=fac*pl
           elso=MATMUL(pl,eps); eload=MATMUL(elso,bee)
           bload=bload+eload*det*weights(i)
         END IF
!-----------------------update the Gauss Point stresses-------------------
         IF(converged.OR.iters==limit)THEN
           tensor(:,i,iel)=tensor(:,i,iel)+sigma-elso
           rload=rload+MATMUL(tensor(:,i,iel),bee)*det*weights(i)
         END IF
       END DO gauss_pts_2
!-----------------------compute the total bodyloads vector ---------------
       bdylds(g)=bdylds(g)+bload; react(g)=react(g)+rload
       bdylds(0)=zero; react(0)=zero
     END DO elements_3; IF(converged.OR.iters==limit)EXIT
   END DO its; totd=totd+loads; pr=zero; ot=zero; pav=zero
   DO i=1,fixed_freedoms
     pr=pr+react(no(i)); ot=ot+react(no(i))*g_coord(2,node(i))
   END DO
   DO i=1,4
     pav=pav+(y_coords(i)-y_coords(i+1))*(tensor(1,1,i)+tensor(1,3,i))*pt5
   END DO
   WRITE(11,'(I5,4E12.4,I5)')iy,iy*presc,-pav,pr,ot,iters
   IF(iters==limit)EXIT
 END DO disp_incs
 CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(totd,nf,0.05_iwp,0.5_iwp,g_coord,g_num,14)
STOP
END PROGRAM p64
```

New scalar integers:

`fixed_freedoms`	number of fixed displacements

New scalar reals:

`c`	cohesion
`fac`	measure of yield surface overshoot (f from 6.35)
`fnew`	value of yield function after stress increment
`gamma`	soil unit weight
`k0`	"at rest" earth pressure coefficient, K_o
`ot`	overturning moment
`pav`	earth force based on stress averaging
`pr`	earth force based on nodal reactions
`presc`	wall displacement increment
`pt5`	set to 0.5
`penalty`	set to 1×10^{20}

New dynamic integer arrays:

`no`	fixed freedom numbers vector
`node`	fixed nodes vector
`sense`	sense of freedoms to be fixed vector

New dynamic real arrays:

`elso`	"plastic" stresses
`gc`	integrating point coordinates
`pl`	plastic $[\mathbf{D}^p]$ matrix
`react`	global nodal reaction forces
`rload`	element nodal reaction forces
`storkv`	holds augmented stiffness diagonal terms

The initial stress method of stress redistribution is demonstrated in a problem of passive earth pressure, in which a smooth wall is translated into a bed of "sand". As in Program 6.1, a rectangular mesh of 8-noded elements is generated with nodes and freedoms counted in the y-direction. An additional feature of this program which appears in the element integration and assembly section is the generation of starting self-weight "at rest" stresses. The coordinates of each Gauss point are calculated using the isoparametric property,

$$x = \sum_{i=1}^{8} N_i \, x_i$$

$$y = \sum_{i=1}^{8} N_i \, y_i \tag{6.40}$$

with the x and y coordinates that result held in the one-dimensional array gc(1) and gc(2) respectively. Only the y coordinate is required in this case, and the vertical stress σ_y is obtained after multiplication by the unit weight held in gamma. The horizontal effective stresses σ_x and σ_z are obtained by multiplying σ_y by the "at rest" earth pressure coefficient K_o held in k0.

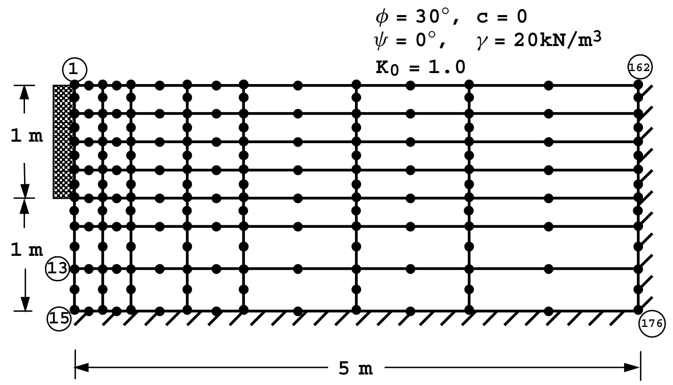
Data relating to the geometry and boundary conditions follow a familiar course. After the stiffness matrix formulation, the nodes and senses of the freedoms, which are to receive prescribed displacements are read, followed by the plastic convergence tolerance tol, the iteration ceiling limit, the number of constant displacement increments that are to be applied incs and the magnitude of the displacement increment held in presc. It may be noted that the iteration ceiling does not need to be as high as when using load control. Convergence is quicker when using displacement control, especially as failure conditions are approached, since unconfined flow cannot occur. The "penalty" technique is used to implement the prescribed displacements, as described in Section 3.6.

The program follows familiar lines until the calculation of the failure function. Initially, the failure function fnew is obtained after adding the full elastic stress increment to those stresses existing previously. If fnew is positive, indicating a yielding Gauss point, then the failure function f is obtained using just those stresses existing previously. The scaling parameter fac is then calculated as described in equation (6.35). The plastic stress–strain matrix $[\mathbf{D}^p]$ for a Mohr–Coulomb material is formed by the subroutine mcdpl (if implementing the von Mises criterion, the subroutine vmdpl should be substituted) using stresses that have been factored to ensure they lie on the failure surface. The resulting matrix pl is multiplied by the scaling parameter fac and then by the total strain increment array eps to yield the "plastic" stress increment array elso. This is simple "forward Euler" integration of the "rate" equations. "Implicit" versions are described in the next sections. Integrals of the type described by equation (6.34) then follow and the array bdylds is accumulated from each element. It may be noted that in the algorithm presented here, the body loads vector is completely reformed at each iteration. This is in contrast to the viscoplasticity algorithm presented in Programs 6.1, 6.2, and 6.3, in which the body loads vector was accumulated at each iteration.

At convergence, the stresses must be updated ready for the next displacement (load) increment. This involves adding, to the stresses remaining from the previous increment, the one-dimensional array of total stress increments sigma minus the one-dimensional array of corrective "plastic" stresses elso.

The example problem shown in Figure 6.19 represents a "sand" with strength parameters $\phi = 30°$, $c = 0$, and dilation angle $\psi = 30°$, subjected to prescribed displacements along the left face. The displacement increments are applied to the x-components of displacement at the nine nodes adjacent to the hypothetical smooth, rigid wall shown hatched. The initial stresses in the ground are calculated assuming the unit weight $\gamma = 20$ kN/m^3 and "at rest" earth pressure coefficient $K_o = 1$.

Following each displacement increment, and after numerical convergence, the resultant force on the wall is calculated in two ways. Firstly, the force on the wall is computed by averaging the σ_x stresses at the eight Gauss points closest to the wall, and this result is held in pav. Secondly, the nodal reactions are back-figured from the converged stress



```
nxe  nye  np_types
7    7    1

prop(phi,c,psi,gamma,ko,e,v)
30.0  0.0  30.0  20.0  1.0  1.0e5  0.3

etype(not needed)

x_coords, y_coords
0.0   0.25   0.5   1.0   1.5   2.5   3.5    5.0
1.0   0.75   0.5   0.25  0.0  -0.25 -0.625 -1.0

nr,(k,nf(:,k),i=1,nr)
29
 15 0 0   23 0 0   38 0 0   46 0 0   61 0 0   69 0 0
 84 0 0   92 0 0  107 0 0  115 0 0  130 0 0  138 0 0
153 0 0  161 0 0  162 0 0  163 0 0  164 0 0  165 0 0
166 0 0  167 0 0  168 0 0  169 0 0  170 0 0  171 0 0
172 0 0  173 0 0  174 0 0  175 0 0  176 0 0

fixed_freedoms,(node(i),sense(i),i=1,fixed_freedoms)
9
1 1   2 1   3 1   4 1   5 1   6 1
7 1   8 1   9 1

tol    limit  incs  presc
0.001  75     35    2.0e-5
```

Figure 6.19 Mesh and data for Program 6.4 example

field using,

$$\{\mathbf{P}\}_r = \sum_{elements}^{all} \iint [\mathbf{B}]^{\mathrm{T}} \{\sigma\}\, \mathrm{d}x\, \mathrm{d}y \tag{6.41}$$

and this is held in `pr`. By multiplying the nodal reaction forces about their distance from the base of the wall, the overturning moment can also be estimated, held in `ot`.

The output shown in Figure 6.20 gives the step number, the accumulated wall displacement, the resultant force (averaging and reactions), the overturning moment and the number of iterations to convergence. These results are plotted in Figure 6.21 and show that

```
 There are    294 equations and the skyline storage is  10521

  step   disp       load(av)     load(react)   moment     iters
    1  0.2000E-04  0.1097E+02  0.1197E+02  0.3678E+01    2
    2  0.4000E-04  0.1194E+02  0.1311E+02  0.4023E+01    2
    3  0.6000E-04  0.1292E+02  0.1425E+02  0.4368E+01    4
    4  0.8000E-04  0.1385E+02  0.1535E+02  0.4676E+01   31
    5  0.1000E-03  0.1477E+02  0.1644E+02  0.4971E+01   19
    6  0.1200E-03  0.1569E+02  0.1752E+02  0.5262E+01   10
    7  0.1400E-03  0.1660E+02  0.1860E+02  0.5548E+01   14
    8  0.1600E-03  0.1751E+02  0.1968E+02  0.5833E+01    8
    9  0.1800E-03  0.1842E+02  0.2076E+02  0.6115E+01   13
   10  0.2000E-03  0.1933E+02  0.2183E+02  0.6397E+01    4
   11  0.2200E-03  0.2021E+02  0.2288E+02  0.6656E+01   32
   12  0.2400E-03  0.2107E+02  0.2392E+02  0.6904E+01   23
   13  0.2600E-03  0.2193E+02  0.2495E+02  0.7148E+01   13
   14  0.2800E-03  0.2279E+02  0.2597E+02  0.7384E+01   22
   15  0.3000E-03  0.2363E+02  0.2698E+02  0.7614E+01   16
   16  0.3200E-03  0.2444E+02  0.2796E+02  0.7819E+01   30
   17  0.3400E-03  0.2523E+02  0.2884E+02  0.8004E+01   32
   18  0.3600E-03  0.2601E+02  0.2970E+02  0.8186E+01   13
   19  0.3800E-03  0.2675E+02  0.3048E+02  0.8355E+01   40
   20  0.4000E-03  0.2744E+02  0.3120E+02  0.8519E+01   44
   21  0.4200E-03  0.2810E+02  0.3187E+02  0.8676E+01   28
   22  0.4400E-03  0.2866E+02  0.3243E+02  0.8815E+01   40
   23  0.4600E-03  0.2918E+02  0.3293E+02  0.8945E+01   29
   24  0.4800E-03  0.2961E+02  0.3330E+02  0.9068E+01   64
   25  0.5000E-03  0.3001E+02  0.3363E+02  0.9180E+01   29
   26  0.5200E-03  0.3029E+02  0.3387E+02  0.9264E+01   44
   27  0.5400E-03  0.3043E+02  0.3403E+02  0.9313E+01   45
   28  0.5600E-03  0.3056E+02  0.3416E+02  0.9353E+01   18
   29  0.5800E-03  0.3067E+02  0.3427E+02  0.9387E+01   11
   30  0.6000E-03  0.3076E+02  0.3437E+02  0.9414E+01   13
   31  0.6200E-03  0.3085E+02  0.3447E+02  0.9439E+01    7
   32  0.6400E-03  0.3092E+02  0.3455E+02  0.9459E+01   11
   33  0.6600E-03  0.3099E+02  0.3463E+02  0.9474E+01   14
   34  0.6800E-03  0.3105E+02  0.3468E+02  0.9486E+01   12
   35  0.7000E-03  0.3110E+02  0.3474E+02  0.9497E+01    3
```

Figure 6.20 Results from Program 6.4 example

the force builds up to a maximum value of around 31 kN/m when using average stresses. This is in close agreement with the closed form Rankine solution of 30 kN/m, despite the relatively crude mesh. The higher result obtained by nodal reactions is commonly observed in analyses of this type, due in part to the high shear stress concentration at the bottom edge of the wall. The displacement vectors of the mesh corresponding to passive failure of the soil behind the wall are shown in Figure 6.22. The Rankine passive mechanism outcropping at an angle of 30° to the horizontal is reproduced.

The initial stress algorithm presented in this program will tend to overestimate collapse loads, especially if the displacement (load) steps are made too big. Users are recommended to try one or two different increment sizes to test the sensitivity of the solutions. The problem is caused by incremental "drift" of the stress state at individual Gauss points into illegal stress space, in spite of apparent numerical convergence. Although not included in the present work, various strategies are available (e.g. Nayak and Zienkiewicz, 1972) for drift correction. In the next section, more complicated "stress-return" procedures are illustrated which ensure stresses at each Gauss point return accurately to the yield surface.

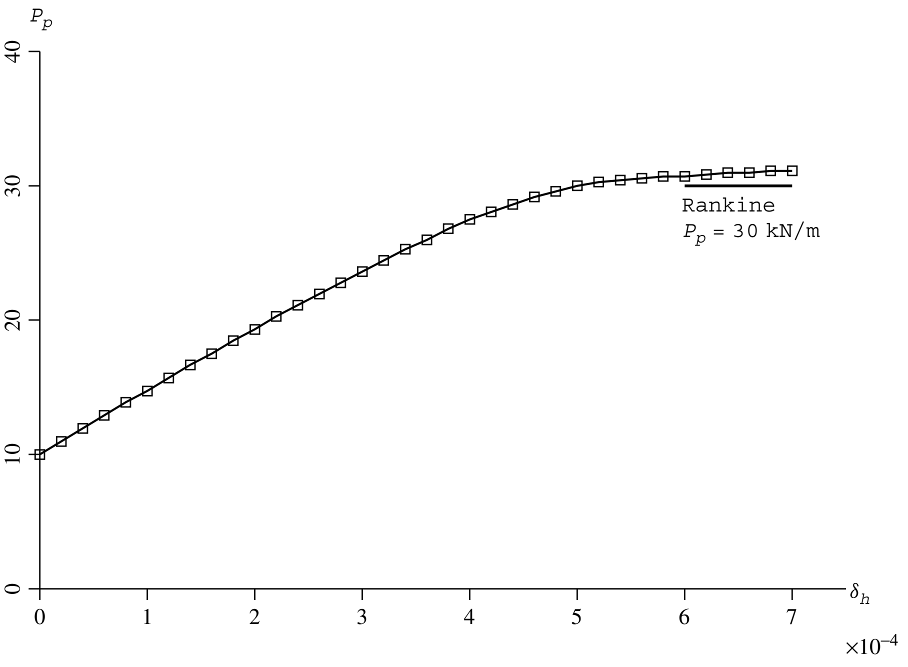


Figure 6.21 Passive force (based on stress averaging) versus horizontal displacement from Program 6.4 example

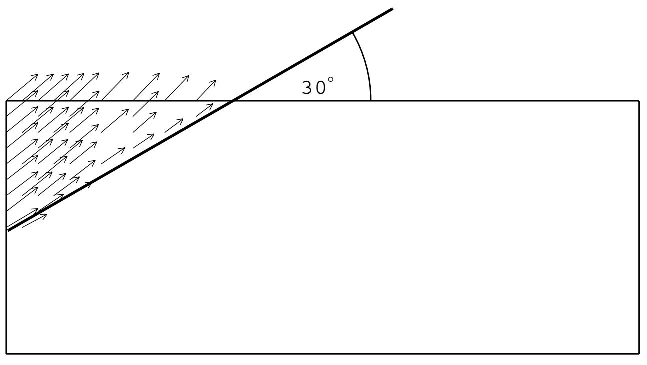


Figure 6.22 Displacement vectors at passive failure from Program 6.4 example

6.9 Elasto-plastic rate integration

For the purposes of this description, we return to elastic-perfectly plastic materials obeying the von Mises failure criterion. Similar, if more complicated arguments apply to Mohr–Coulomb materials.

Using the notation previously developed in Sections 6.3 and 6.4, if F is the yield function and J_2 the second invariant of the deviatoric stress tensor, from (6.7),

$$\begin{aligned} F &= \overline{\sigma} - \sqrt{3}c_u \\ &= \sqrt{3J_2} - \sqrt{3}c_u \end{aligned} \tag{6.42}$$

where

$$J_2 = \frac{t^2}{2} = \frac{1}{6}\left[(\sigma_x - \sigma_y)^2 + (\sigma_y - \sigma_z)^2 + (\sigma_z - \sigma_x)^2 + 6\tau_{xy}^2 + 6\tau_{yz}^2 + 6\tau_{zx}^2\right] \tag{6.43}$$

The first derivative of F with respect to the stresses is

$$\left\{\frac{\partial F}{\partial \boldsymbol{\sigma}}\right\} = \{\mathbf{a}\} = \frac{\partial F}{\partial J_2}\left\{\frac{\partial J_2}{\partial \boldsymbol{\sigma}}\right\} \tag{6.44}$$

which can be written as,

$$\{\mathbf{a}\} = \frac{1.5}{\sqrt{3J_2}}\left\{\begin{array}{c} s_x \\ s_y \\ s_z \\ 2\tau_{xy} \\ 2\tau_{yz} \\ 2\tau_{zx} \end{array}\right\} \tag{6.45}$$

where s_x and so on, represent the deviatoric components from equations (6.3).

The second derivative of F with respect to stress is

$$\left[\frac{\partial^2 F}{\partial \boldsymbol{\sigma}^2}\right] = \left[\frac{\partial \mathbf{a}}{\partial \boldsymbol{\sigma}}\right] = \frac{1}{2\sqrt{3J_2}}[\mathbf{A}] - \frac{1}{\sqrt{3J_2}}\{\mathbf{a}\}\{\mathbf{a}\}^{\mathrm{T}} \tag{6.46}$$

where

$$[\mathbf{A}] = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \end{bmatrix} \tag{6.47}$$

Ortiz and Popov (1985) described various methods of elasto-plastic rate integration, which essentially consist of an (elastic) predictor, followed by a plastic corrector to ensure the final stress is (nearly) on the yield surface.

Referring to Figure 6.23 let $\{\boldsymbol{\sigma}_X\}$ refer to the unyielded stress at the start of a step and $\left\{\boldsymbol{\Delta\sigma}^e\right\}$ the (elastic) increment. The stress crosses the yield surface at $\{\boldsymbol{\sigma}_A\}$ while the elastic increment ends up at $\{\boldsymbol{\sigma}_B\}$ We wish to return to the "correct" stresses on the yield surface at $\{\boldsymbol{\sigma}_C\}$.

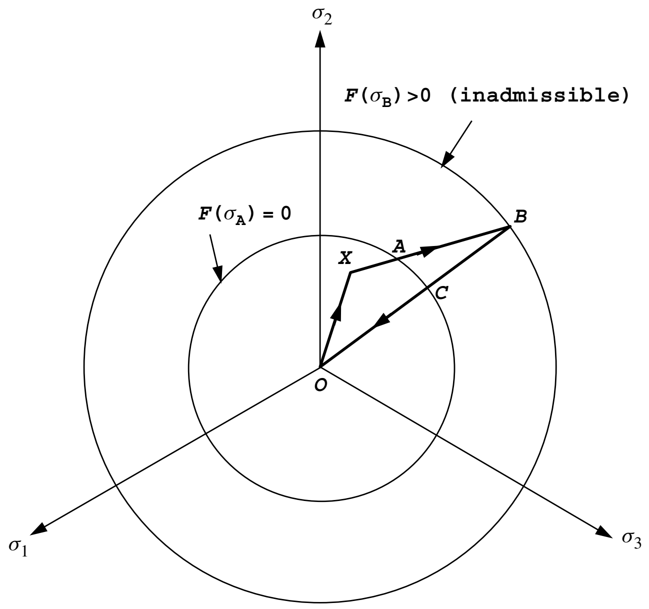


Figure 6.23 Stress correction

If $\{\boldsymbol{\Delta\epsilon}\}$ is the total incremental strain, $\left\{\boldsymbol{\Delta\epsilon}^p\right\}$ the incremental plastic strain, and λ the scalar multiplier (6.31) an elastic stress–strain matrix $[\mathbf{D}^e]$ will lead to

$$\{\boldsymbol{\sigma}_C\} = \{\boldsymbol{\sigma}_A\} + [\mathbf{D}^e]\left(\{\boldsymbol{\Delta\epsilon}\} - \left\{\boldsymbol{\Delta\epsilon}^p\right\}\right) \tag{6.48}$$

where

$$\left\{\boldsymbol{\Delta\epsilon}^p\right\} = \lambda\,\{\mathbf{a}\} \tag{6.49}$$

The derivative $\{\mathbf{a}\}$ in the above equation is evaluated at $(1-\beta)\{\boldsymbol{\sigma}_A\} + \beta\{\boldsymbol{\sigma}_C\}$, and the scalar interpolating parameter β (similar to θ in Chapters 8 and 9) can vary in the range $0 \leq \beta \leq 1$.

6.9.1 Forward Euler method

Here $\beta = 0$ and the rate equation is integrated at the point at which the yield surface is crossed ($\{\boldsymbol{\sigma}_A\}$ in Figure 6.23). Thus

$$F\left(\{\boldsymbol{\sigma}_X\} + \alpha\left\{\boldsymbol{\Delta\sigma}^e\right\}\right) = 0 \tag{6.50}$$

For a non-hardening von Mises material the point can be found explicitly from:

$$\alpha = \frac{-3B \pm \sqrt{9B^2 - 12A(3C - 3c_u^2)}}{6A} \tag{6.51}$$

where

$$\begin{aligned} A &= \frac{1}{2}((\Delta s_x^e)^2 + (\Delta s_y^e)^2 + (\Delta s_z^e)^2) + (\Delta \tau_{xy}^e)^2 \\ B &= s_x \Delta s_x^e + s_y \Delta s_y^e + s_z \Delta s_z^e + 2\Delta \tau_{xy} \tau_{xy}^e \\ C &= \frac{1}{2}(s_x^2 + s_y^2 + s_z^2) + \tau_{xy}^2 \end{aligned} \tag{6.52}$$

but an approximate α can be found by linearly interpolating between points X and B from,

$$\alpha \approx \frac{-F(\{\boldsymbol{\sigma}_X\})}{F(\{\boldsymbol{\sigma}_B\}) - F(\{\boldsymbol{\sigma}_X\})} \tag{6.53}$$

The remaining stress $(1-\alpha)[\mathbf{D}^e]\{\boldsymbol{\Delta\epsilon}\}$ causes the "illegal" stress state outside the yield surface. For non-hardening plasticity, it is assumed that once a stress state reaches a yield surface, subsequent changes in stress may shift the stress state to a different position on the yield surface but not outside it (6.30), hence

$$\Delta F = \{\mathbf{a}\}^{\mathrm{T}} \{\boldsymbol{\Delta\sigma}\} = 0 \tag{6.54}$$

thus

$$\Delta F = \{\mathbf{a}\}^{\mathrm{T}} ([\mathbf{D}^e]\{\boldsymbol{\Delta\epsilon}\} - (1-\alpha)\lambda[\mathbf{D}^e]\{\mathbf{a}\}) = 0 \tag{6.55}$$

Assuming that the derivative vector $\{\mathbf{a}\}$ as evaluated at point A is called $\{\mathbf{a}_A\}$, the following expression for λ_A, the "plastic multiplier" at A is given by,

$$\lambda_A = \frac{\{\mathbf{a}_A\}^{\mathrm{T}}\ [\mathbf{D}^e]\ \{\boldsymbol{\Delta\epsilon}\}}{(1-\alpha)\ \{\mathbf{a}_A\}^{\mathrm{T}}\ [\mathbf{D}^e]\ \{\mathbf{a}_A\}} \tag{6.56}$$

The final stress is then

$$\{\boldsymbol{\sigma}_C\} = \{\boldsymbol{\sigma}_X\} + \left\{\boldsymbol{\Delta\sigma}^e\right\} - \lambda_A[\mathbf{D}^e]\ \{\mathbf{a}_A\} \tag{6.57}$$

This is the method used in equation (6.35) in which $f = 1 - \alpha$.

6.9.2 Backward Euler method

Here the rate equation is integrated at the "illegal" state B ($\beta = 1$). This results in a simple evaluation of the plastic multiplier λ for non-hardening von Mises materials. A first order Taylor expansion of the yield function at B gives:

$$F(\{\boldsymbol{\sigma}_C\}) = F(\{\boldsymbol{\sigma}_B\}) + \left\{\frac{\partial F}{\partial \boldsymbol{\sigma}}\right\}^{\mathrm{T}} \{\boldsymbol{\Delta\sigma}\} \tag{6.58}$$

By enforcing consistency of the yield function at point C,

$$0 = F(\{\boldsymbol{\sigma}_B\}) - \lambda_B \{\mathbf{a}_B\}^{\mathrm{T}} [\mathbf{D}^e] \{\mathbf{a}_B\} \tag{6.59}$$

so that

$$\lambda_B = \frac{F(\{\boldsymbol{\sigma}_B\})}{\{\mathbf{a}_B\}^{\mathrm{T}}[\mathbf{D}^e]\{\mathbf{a}_B\}} \tag{6.60}$$

The change in stress is given by,

$$\{\boldsymbol{\Delta\sigma}\} = \left\{\boldsymbol{\Delta\sigma}^e\right\} - \frac{F(\{\boldsymbol{\sigma}_B\})[\mathbf{D}^e]\{\mathbf{a}_B\}}{\{\mathbf{a}_B\}^{\mathrm{T}}[\mathbf{D}^e]\{\mathbf{a}_B\}} \tag{6.61}$$

and final stress by,

$$\{\boldsymbol{\sigma}\} = \{\boldsymbol{\sigma}_X\} + \left\{\boldsymbol{\Delta\sigma}^e\right\} - \lambda_B[\mathbf{D}^e]\ \{\mathbf{a}_B\} \tag{6.62}$$

Rice and Tracey (1973) advocated a mean normal method for a von Mises yield criterion so that

$$\frac{\{\mathbf{a}_A + \mathbf{a}_B\}^{\mathrm{T}}\left\{\boldsymbol{\Delta\sigma}^e\right\}}{2} = 0 \tag{6.63}$$

In a von Mises yield criterion under plane strain and 3D stress states, the yield surface appears as a circle on the deviatoric plane. Any “illegal” stress can be corrected along a radial path directed from the hydrostatic stress axis. The final deviatoric stress at point C is

$$\{\mathbf{s}_C\} = \frac{\sqrt{3c_u}}{\sqrt{3J_2}}\{\mathbf{s}_B\} \tag{6.64}$$

and the components of $\{\boldsymbol{\sigma}_C\}$ can then be determined by superimposing the hydrostatic stress from point B.

In practice, it has been found that this method offers no advantages over forward Euler in constant stiffness algorithms. The same is not true for tangent stiffness methods as is shown in the next paragraph.

6.10 Tangent stiffness approaches

The difference between constant stiffness and tangent stiffness methods was discussed in Section 6.1. In general, constant stiffness methods can be attractive in displacement-controlled situations (see Figure 6.20 where the number of iterations per displacement increment is modest), but in load-controlled situations, particularly close to collapse, large numbers of iterations tend to arise (see e.g. Figure 6.10). Tangent stiffness approaches require fewer iterations per load step, however this saving is counterbalanced by the speed of constant stiffness methods, in which the global stiffness matrix is only factorised once. If convergence in cases like Figure 6.10 is monitored, it will be found that most Gauss points have converged to the yield surface, leaving only a few Gauss points responsible for the lack of convergence. Tangent stiffness methods, with backward Euler integration can significantly improve the convergence properties of algorithms to the point where the cost of re-forming and re-factorising the global stiffness can be justified.

6.10.1 Inconsistent tangent matrix

The change in stress is composed of two parts, the elastic predictor $[\mathbf{D}^e]\{\boldsymbol{\Delta\epsilon}\}$ and a plastic corrector $\lambda[\mathbf{D}^e]\{\mathbf{a}\}$, that is

$$\{\boldsymbol{\Delta\sigma}\} = [\mathbf{D}^e](\{\boldsymbol{\Delta\epsilon}\} - \lambda\{\mathbf{a}\}) \tag{6.65}$$

Substituting λ from (6.56) ($\alpha = 0$) into the above equation gives,

$$\{\boldsymbol{\Delta\sigma}\} = [\mathbf{D}^e]\left(\{\boldsymbol{\Delta\epsilon}\} - \frac{\{\mathbf{a}\}^{\mathrm{T}}[\mathbf{D}^e]\{\boldsymbol{\Delta\epsilon}\}}{\{\mathbf{a}\}^{\mathrm{T}}[\mathbf{D}^e]\{\mathbf{a}\}}\{\mathbf{a}\}\right) \tag{6.66}$$

and hence

$$\{\boldsymbol{\Delta\sigma}\} = \left([\mathbf{D}^e] - \frac{[\mathbf{D}^e]\{\mathbf{a}\}\{\mathbf{a}\}^{\mathrm{T}}[\mathbf{D}^e]}{\{\mathbf{a}\}^{\mathrm{T}}[\mathbf{D}^e]\{\mathbf{a}\}}\right)\{\boldsymbol{\Delta\epsilon}\} \tag{6.67}$$

or

$$\{\boldsymbol{\Delta\sigma}\} = [\mathbf{D}^{ep}]\{\boldsymbol{\Delta\epsilon}\} \tag{6.68}$$

where $[\mathbf{D}^{ep}]$ is known as the standard or "inconsistent" tangent matrix.

6.10.2 Consistent tangent matrix

With the backward Euler integration scheme, a consistent tangent modular matrix can be formed.

$$\begin{aligned}\{\boldsymbol{\sigma}\} &= \{\boldsymbol{\sigma}_B\} - \lambda_B[\mathbf{D}^e]\ \{\mathbf{a}_B\}\\ &= (\{\boldsymbol{\sigma}_X\} + [\mathbf{D}^e]\{\boldsymbol{\Delta\epsilon}\}) - \lambda_B[\mathbf{D}^e]\ \{\mathbf{a}_B\}\end{aligned} \tag{6.69}$$

On differentiation we get,

$$\{\boldsymbol{\Delta\sigma}\} = [\mathbf{D}^e]\{\boldsymbol{\Delta\epsilon}\} - \Delta\lambda[\mathbf{D}^e]\{\mathbf{a}\} - \lambda_B[\mathbf{D}^e]\left[\left(\frac{\partial\mathbf{a}}{\partial\boldsymbol{\sigma}}\right)_B\right]\{\boldsymbol{\Delta\sigma}\} \tag{6.70}$$

or

$$\{\boldsymbol{\Delta\sigma}\} = \left[[\mathbf{I}] + \lambda_B[\mathbf{D}^e]\left[\left(\frac{\partial\mathbf{a}}{\partial\boldsymbol{\sigma}}\right)_B\right]\right]^{-1}[\mathbf{D}^e](\{\boldsymbol{\Delta\epsilon}\} - \Delta\lambda\{\mathbf{a}\}) \tag{6.71}$$

$$= [\mathbf{R}](\{\boldsymbol{\Delta\epsilon}\} - \Delta\lambda\{\mathbf{a}\}) \tag{6.72}$$

and hence

$$\{\boldsymbol{\Delta\sigma}\} = \left([\mathbf{R}] - \frac{[\mathbf{R}]\{\mathbf{a}\}\{\mathbf{a}\}^{\mathrm{T}}[\mathbf{R}]}{\{\mathbf{a}\}^{\mathrm{T}}[\mathbf{R}]\{\mathbf{a}\}}\right)\{\boldsymbol{\Delta\epsilon}\} \tag{6.73}$$

or

$$\{\boldsymbol{\Delta\sigma}\} = [\mathbf{D}^{epc}]\{\boldsymbol{\Delta\epsilon}\} \tag{6.74}$$

where $[\mathbf{D}^{epc}]$ is known as the "consistent" tangent matrix.

6.10.3 Convergence criterion

Programs 6.1 to 6.4 used a very simple convergence criterion that was essentially based on the body loads vector `bdylds` used to correct the out-of-balance stresses. In the programs the nodal displacements (usually called `loads`) caused by the `bdylds` at successive iterations were compared. Convergence was said to have occurred, if the absolute change in all the components of `loads`, as a fraction of the maximum absolute component of `loads`, was less than a tolerance `tol` of say, 0.001.

When the convergence of `loads` is examined, it is found that in typical problems nearly all Gauss Points converge early, and most of the time is taken in forcing the stresses at a few points towards the yield surface.

In the consistent tangent method, the efficiency of the return algorithm is such that all Gauss Points converge much faster to the yield surface. On the other hand, there can be no concept of a converging `bdylds`. Rather, the residual remaining in `bdylds` tends to zero as convergence is approached (this is actually how many codes operate for the constant stiffness, forward Euler, case as well). A criterion based on the size of the maximum component of the reducing `bdylds` as a percentage of, say, the RMS of `bdylds` can be used.

In practice, when an element assembly technique is chosen, the strategy for constant stiffness is simple to implement and just as efficient computationally as the consistent tangent approach. This is because the plastic iterations involve only forward and back-substitutions in a direct equation-solving process.

When a tangent stiffness method is used, the extra time involved in reforming the stiffness matrices and completely re-solving the equilibrium equations can more than compensate for the reduced iteration counts.

However, when iterative strategies are adopted for equilibrium equation solution (e.g. Program 6.2) the equilibrium equations have to be “reassembled” and solved on every iteration anyway. In these circumstances, the consistent tangent stiffness with backward Euler return, leading to low iteration counts, is essential.

Program 6.5 Plane strain bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Initial stress method. Tangent stiffness. Consistent return algorithm.

```
PROGRAM p65
!-------------------------------------------------------------------------
! Program 6.5 Plane strain bearing capacity analysis of an elastic-plastic
!             (von Mises) material using 8-node rectangular
!             quadrilaterals. Initial stress method. Tangent stiffness.
!             Consistent return algorithm.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,incs,iters,iy,k,limit,loaded_nodes,ndim=2,ndof=16,nels,   &
   neq,nip=4,nn,nod=8,nodof=2,nprops=3,np_types,nr,nst=4,nxe,nye
 REAL(iwp)::bot,det,dlam,dsbar,dslam,d3=3.0_iwp,ff,fftol,fnew,fstiff,     &
   lode_theta,ltol,one=1.0_iwp,ptot,sigm,tloads,tol,top,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
```

```
   no(:),num(:)
 REAL(iwp),ALLOCATABLE::acat(:,:),acatc(:,:),bee(:,:),bdylds(:),bload(:), &
   caflow(:),coord(:,:),daatd(:,:),ddylds(:),dee(:,:),der(:,:),deriv(:,:),&
   dl(:,:),dload(:),dsigma(:),eld(:),eload(:),elso(:),eps(:),g_coord(:,:),&
   jac(:,:),km(:,:),kv(:),loads(:),points(:,:),prop(:,:),qinc(:),qinva(:),&
   qinvr(:),qmat(:,:),ress(:),rmat(:,:),sigma(:),stress(:),tensor(:,:,:), &
   totd(:),val(:,:),vmfl(:),vmfla(:),vmflq(:),vmtemp(:,:),weights(:),     &
   x_coords(:),y_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,fftol,ltol,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),                 &
   tensor(nst,nip,nels),g_g(ndof,nels),coord(nod,ndim),jac(ndim,ndim),    &
   der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),           &
   km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof),eload(ndof),   &
   elso(nst),g(ndof),vmfl(nst),qinvr(nst),dl(nip,nels),stress(nst),       &
   dload(ndof),caflow(nst),dsigma(nst),ress(nst),rmat(nst,nst),           &
   acat(nst,nst),acatc(nst,nst),qmat(nst,nst),qinva(nst),daatd(nst,nst),  &
   vmflq(nst),vmfla(nst),vmtemp(1,nst),prop(nprops,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),totd(0:neq),ddylds(0:neq))
!-----------------------loop the elements to find global arrays sizes-----
 kdiag=0
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I7))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); kv=zero
!--------------starting element stiffness integration and assembly--------
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!-----------------------read load weightings and factorise equations------
 READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(no(i),val(i,:),i=1,loaded_nodes); CALL sparin(kv,kdiag)
!-----------------------load increment loop-------------------------------
 READ(10,*)tol,limit,incs; ALLOCATE(qinc(incs)); READ(10,*)qinc
 WRITE(11,'(/A)')"  step   load        disp      iters"
 totd=zero; tensor=zero; dl=zero; ptot=zero
 load_increments: DO iy=1,incs
   ptot=ptot+qinc(iy); bdylds=zero; loads=zero; iters=0
   DO i=1,loaded_nodes; loads(nf(:,no(i)))=val(i,:)*qinc(iy); END DO
!-----------------------plastic iteration loop----------------------------
```

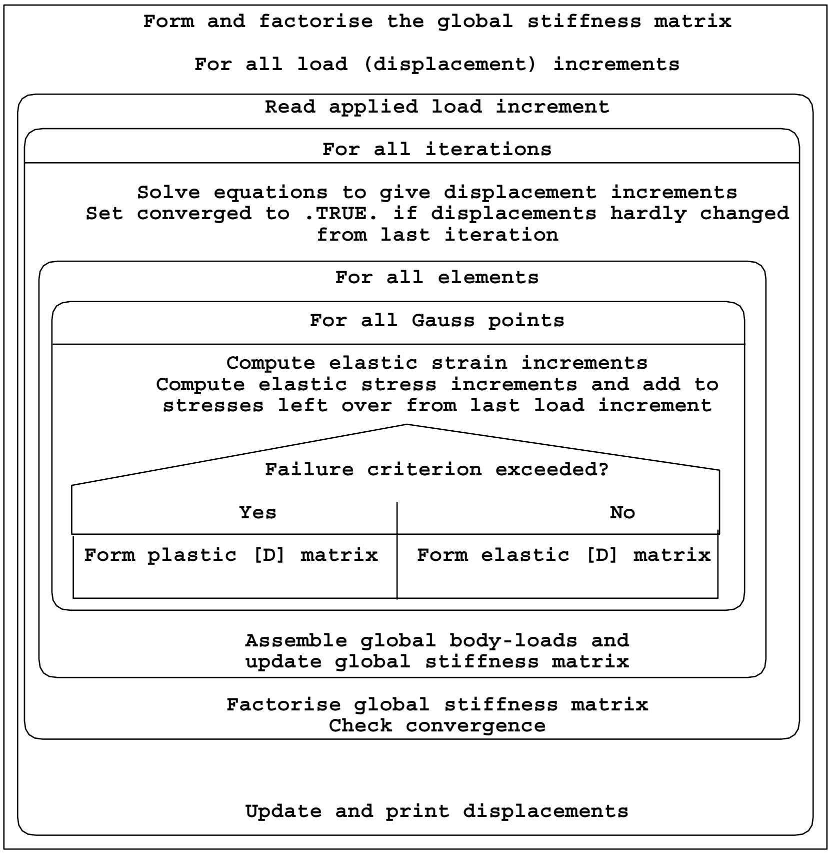
```
   plastic_iters: DO
     iters=iters+1; IF(iters/=1)loads=zero
     WRITE(*,'(A,F8.2,A,I4)')"  load",ptot,"  iteration",iters
     loads=loads+bdylds; CALL spabac(kv,loads,kdiag)
     bdylds=zero; ddylds=zero; kv=zero
!-----------------------go round the Gauss Points -----------------------
     elements_3: DO iel=1,nels
       bload=zero; dload=zero; num=g_num(:,iel)
       coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g); km=zero
       gauss_pts_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
         CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
         stress=tensor(:,i,iel); CALL invar(stress,sigm,dsbar,lode_theta)
         ff=dsbar-SQRT(d3)*prop(1,etype(iel))
         IF(ff>fftol)THEN
           dlam=dl(i,iel); CALL vmflow(stress,dsbar,vmfl)
           CALL fmrmat(vmfl,dsbar,dlam,dee,rmat); caflow=MATMUL(rmat,vmfl)
           bot=DOT_PRODUCT(vmfl,caflow); CALL formaa(vmfl,rmat,daatd)
           dee=rmat-daatd/bot
         END IF
         sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
         CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         fnew=dsbar-SQRT(d3)*prop(1,etype(iel)); fstiff=fnew; elso=zero
         IF(fnew>=zero)THEN
           CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
           CALL vmflow(stress,dsbar,vmfl); caflow=MATMUL(dee,vmfl)
           bot=DOT_PRODUCT(vmfl,caflow); dlam=fnew/bot; elso=caflow*dlam
           stress=tensor(:,i,iel)+sigma-elso
           CALL invar(stress,sigm,dsbar,lode_theta)
           fnew=dsbar-SQRT(d3)*prop(1,etype(iel))
           iterate_on_fnew: DO
             CALL vmflow(stress,dsbar,vmfl); caflow=MATMUL(dee,vmfl)*dlam
             ress=stress-(tensor(:,i,iel)+sigma-caflow)
             CALL fmacat(vmfl,acat); acat=acat/dsbar
             acatc=MATMUL(dee,acat); qmat=acatc*dlam
             DO k=1,4; qmat(k,k)=qmat(k,k)+one; END DO; CALL invert(qmat)
             vmtemp(1,:)=vmfl; vmtemp=MATMUL(vmtemp,qmat)
             vmflq=vmtemp(1,:); top=DOT_PRODUCT(vmflq,ress)
             vmtemp=MATMUL(vmtemp,dee); vmfla=vmtemp(1,:)
             bot=DOT_PRODUCT(vmfla,vmfl); dslam=(fnew-top)/bot
             qinvr=MATMUL(qmat,ress); qinva=MATMUL(MATMUL(qmat,dee),vmfl)
             dsigma=-qinvr-qinva*dslam; stress=stress+dsigma
             CALL invar(stress,sigm,dsbar,lode_theta)
             fnew=dsbar-SQRT(d3)*prop(1,etype(iel)); dlam=dlam+dslam
             IF(fnew<tol)EXIT
           END DO iterate_on_fnew
           dl(i,iel)=dlam; elso=tensor(:,i,iel)+sigma-stress
           eload=MATMUL(elso,bee); bload=bload+eload*det*weights(i)
           CALL vmflow(stress,dsbar,vmfl)
           CALL fmrmat(vmfl,dsbar,dlam,dee,rmat); caflow=MATMUL(rmat,vmfl)
           bot=DOT_PRODUCT(vmfl,caflow); CALL formaa(vmfl,rmat,daatd)
           dee=rmat-daatd/bot
         END IF
         IF(fstiff<zero)                                                &
           CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
         km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!------------------------update the Gauss Point stresses-------------------
         tensor(:,i,iel)=tensor(:,i,iel)+sigma-elso; stress=tensor(:,i,iel)
         eload=MATMUL(stress,bee); dload=dload+eload*det*weights(i)
       END DO gauss_pts_2
!------------------------compute the total bodyloads vector----------------
       bdylds(g)=bdylds(g)+bload; bdylds(0)=zero; ddylds(g)=ddylds(g)+dload
       ddylds(0)=zero; CALL fsparv(kv,km,g,kdiag)
     END DO elements_3; CALL sparin(kv,kdiag); tloads=SUM(bdylds)
     IF(iters==1)converged=.FALSE.
     IF(iters/=1.AND.tloads<ltol)converged=.TRUE.; totd=totd+loads
     IF(converged.OR.iters==limit)EXIT
   END DO plastic_iters; totd=totd+loads
   WRITE(11,'(I5,2E12.4,I5)')iy,ptot,totd(nf(2,no(1))),iters
   IF(iters==limit)EXIT
 END DO load_increments
 CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(totd,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p65
```

New scalar reals:

`bot`	holds several dot products
`dlam`	plastic multiplier λ
`dslam`	plastic multiplier increment $\Delta\lambda$
`ff`	holds a value of the yield function
`fftol`	tolerance on yield function
`fstiff`	holds a value of the yield function
`ltol`	tolerance on `tloads`
`top`	holds a dot product
`tloads`	holds the sum of `bdylds`

New dynamic real arrays:

`acat`	used in development of (6.74)
`acatc`	used in development of (6.74)
`caflow`	used in development of (6.74)
`daatd`	used in development of (6.74)
`ddylds`	global body loads
`dl`	holds plastic multiplier λ for all Gauss points
`dload`	element body loads
`dsigma`	stress increment
`qinva`	used in development of (6.74)
`qinvr`	used in development of (6.74)
`qmat`	used in development of (6.74)
`ress`	used in development of (6.74)
`rmat`	used in development of (6.74)
`vmfl`	von Mises “flow” vector
`vmfla`	used in development of (6.74)
`vmflq`	used in development of (6.74)
`vmtemp`	used in development of (6.74)



```
Form and factorise the global stiffness matrix
For all load (displacement) increments
    Read applied load increment
    For all iterations
        Solve equations to give displacement increments
        Set converged to .TRUE. if displacements hardly changed
        from last iteration
        For all elements
            For all Gauss points
                Compute elastic strain increments
                Compute elastic stress increments and add to
                stresses left over from last load increment
                Failure criterion exceeded?
                    Yes: Form plastic [D] matrix
                    No:  Form elastic [D] matrix
            Assemble global body-loads and
            update global stiffness matrix
        Factorise global stiffness matrix
        Check convergence
    Update and print displacements
```

Figure 6.24 Structure chart for variable stiffness algorithm with assembly

The structure chart for a typical tangent stiffness approach is shown in Figure 6.24. Preliminary element loops, `elements_1` and `elements_2` set up the geometry and the starting tangent matrix respectively. The load increment loop is entered and new subroutines encountered are `vmflow` which forms the von Mises flow vector from equation (6.45), `fmrmat`, `formaa`, and `fmacat` which are used to form the matrices needed in the development of equation (6.74) at every Gauss point in the `elements_3` loop. The 4×4 matrix `qmat` has to be inverted to complete the return strategy at each Gauss point and ultimately the consistent tangent `dee` matrix is obtained, leading to the consistent `km`.

Figure 6.25 shows the data for the problem analysed, for which we return to the original problem of Figure 6.9. The only new information required is confined to tolerances specific to the tangent stiffness algorithm, namely `fftol=-1.0e-6` and `ltol=5.0e-5`, while the iteration ceiling `limit` can be assigned a much more economic value of, say, 50.

The results are listed as Figure 6.26 and are found to be very similar to those in Figure 6.10 up to the 9th load increment. Up until then, the consistent return algorithm leads to convergence within 4 iterations on every increment. On the final increment (at "failure") Program 6.5 took 28 iterations compared to Program 6.1 which took (at least) 250.

```
nxe  nye   fftol   ltol  np_types
8     4  -1.0e-6  5.0e-5   1

prop(cu,e,v)
100.0  1.0e5  0.3

etype(not needed)

x_coords, y_coords
0.0   1.0   2.0   3.0   4.0   5.5   7.0   9.0  12.0
0.0  -1.25 -2.5  -3.75 -5.0

nr,(k,nf(:,k),i=1,nr)
33
  1 0 1    2 0 1    3 0 1    4 0 1    5 0 1    6 0 1
  7 0 1    8 0 1    9 0 0   14 0 0   23 0 0   28 0 0
 37 0 0   42 0 0   51 0 0   56 0 0   65 0 0   70 0 0
 79 0 0   84 0 0   93 0 0   98 0 0  107 0 0  112 0 0
113 0 0  114 0 0  115 0 0  116 0 0  117 0 0  118 0 0
119 0 0  120 0 0  121 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
5
1   0.0  -0.166667     10  0.0  -0.666667     15  0.0  -0.333333
24  0.0  -0.666667     29  0.0  -0.166667

tol   limit
0.001  50

incs,(qinc(i),i=1,incs)
10
200.0 100.0  50.0  50.0  50.0  30.0  20.0  10.0   5.0   5.0
```

Figure 6.25 Data for Program 6.5 example

```
 There are    184 equations and the skyline storage is   4130

  step   load        disp      iters
    1  0.2000E+03 -0.6592E-02    2
    2  0.3000E+03 -0.1154E-01    4
    3  0.3500E+03 -0.1614E-01    4
    4  0.4000E+03 -0.2285E-01    4
    5  0.4500E+03 -0.3278E-01    4
    6  0.4800E+03 -0.4184E-01    4
    7  0.5000E+03 -0.5031E-01    4
    8  0.5100E+03 -0.5613E-01    4
    9  0.5150E+03 -0.6058E-01    4
   10  0.5200E+03 -0.3307E+00   28
```

Figure 6.26 Results from Program 6.5 example

Program 6.6 Plane strain bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Initial stress method. Tangent stiffness. Consistent return algorithm. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver.

```
PROGRAM p66
!-------------------------------------------------------------------------
! Program 6.6 Plane strain bearing capacity analysis of an elastic-plastic
!             (von Mises) material using 8-node rectangular
!             quadrilaterals. Initial stress method. Tangent stiffness.
!             No global stiffness matrix assembly. Diagonally
!             preconditioned conjugate gradient solver.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,cg_tot,i,iel,incs,iters,iy,k,limit,          &
   loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nn,nod=8,nodof=2,nprops=3,  &
   np_types,nr,nst=4,nxe,nye
 REAL(iwp)::alpha,beta,bot,cg_tol,det,dlam,dsbar,dslam,d3=3.0_iwp,ff,     &
   fftol,fnew,fstiff,lode_theta,ltol,one=1.0_iwp,ptot,sigm,               &
   small_tol=1.e-5_iwp,tloads,tol,top,up,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
 LOGICAL::converged,cg_converged
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::nf(:,:),g(:),no(:),num(:),g_num(:,:),g_g(:,:),      &
   etype(:)
 REAL(iwp),ALLOCATABLE::acat(:,:),acatc(:,:),bee(:,:),bdylds(:),bload(:), &
   caflow(:),coord(:,:),d(:),daatd(:,:),ddylds(:),dee(:,:),der(:,:),      &
   deriv(:,:),diag_precon(:),dl(:,:),dload(:),dsigma(:),eld(:),eload(:),  &
   elso(:),eps(:),g_coord(:,:),jac(:,:),km(:,:),loads(:),p(:),points(:,:),&
   prop(:,:),qinc(:),qinva(:),qinvr(:),qmat(:,:),ress(:),rmat(:,:),       &
   sigma(:),storkm(:,:,:),stress(:),tensor(:,:,:),totd(:),u(:),val(:,:),  &
   vmfl(:),vmfla(:),vmflq(:),vmtemp(:,:),weights(:),x(:),xnew(:),         &
   x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,cg_tol,cg_limit,fftol,ltol,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),                 &
   tensor(nst,nip,nels),g_g(ndof,nels),etype(nels),storkm(ndof,ndof,nels),&
   coord(nod,ndim),stress(nst),dl(nip,nels),jac(ndim,ndim),der(ndim,nod), &
   deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof), &
   eps(nst),sigma(nst),bload(ndof),eload(ndof),elso(nst),g(ndof),         &
   vmfl(nst),qinvr(nst),dload(ndof),caflow(nst),dsigma(nst),ress(nst),    &
   rmat(nst,nst),acat(nst,nst),acatc(nst,nst),qmat(nst,nst),qinva(nst),   &
   daatd(nst,nst),vmflq(nst),vmfla(nst),vmtemp(1,nst),                    &
   prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 WRITE(11,'(A,I6,A)')"There are ",neq,"  equations"
 ALLOCATE(loads(0:neq),bdylds(0:neq),totd(0:neq),ddylds(0:neq),p(0:neq),  &
   x(0:neq),xnew(0:neq),u(0:neq),diag_precon(0:neq),d(0:neq))
!---------------loop the elements to set up element data------------------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 CALL sample(element,points,weights); diag_precon=zero
!----starting element stiffness integration, storage and preconditioner---
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
   END DO gauss_pts_1; storkm(:,:,iel)=km
   DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
 END DO elements_2; diag_precon(1:)=one/diag_precon(1:)
!-----------------------read load weightings------------------------------
 READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(no(i),val(i,:),i=1,loaded_nodes)
!-----------------------load increment loop------------------------------
 READ(10,*)tol,limit,incs; ALLOCATE(qinc(incs)); READ(10,*)qinc
 WRITE(11,'(/A)')                                                       &
 "  step   load        disp      iters    cg iters/plastic iter"
 totd=zero; tensor=zero; xnew=zero; dl=zero; diag_precon(0)=zero; ptot=zero
 load_increments: DO iy=1,incs
   ptot=ptot+qinc(iy);  bdylds=zero; loads=zero; iters=0; cg_tot=0
   DO i=1,loaded_nodes; loads(nf(:,no(i)))=val(i,:)*qinc(iy); END DO
!-----------------------plastic iteration loop  -------------------------
   plastic_iters: DO
     iters=iters+1; IF(iters/=1)loads=zero
     WRITE(*,'(A,F8.2,A,I4)')"  load",ptot,"  iteration",iters
     loads=loads+bdylds
     IF(ABS(SUM(loads))<small_tol)THEN; iters=iters-1; EXIT; END IF
     bdylds=zero; ddylds=zero; d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----------------------pcg equation solution----------------------------
     pcg: DO
       cg_iters=cg_iters+1; u=zero
       elements_3: DO iel=1,nels
         g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
       END DO elements_3
       up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
       loads=loads-u*alpha; d=diag_precon*loads
       beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
       call checon(xnew,x,cg_tol,cg_converged)
       IF(cg_converged.OR.cg_iters==cg_limit)EXIT
     END DO pcg; cg_tot=cg_tot+cg_iters; loads=xnew; loads(0)=zero
     diag_precon=zero
!-----------------------go round the Gauss Points------------------------
     elements_4: DO iel=1,nels
       bload=zero; dload=zero; num=g_num(:,iel)
       coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g); km=zero
       gauss_points_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
         CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
         stress=tensor(:,i,iel); CALL invar(stress,sigm,dsbar,lode_theta)
         ff=dsbar-SQRT(d3)*prop(1,etype(iel))
         IF(ff>fftol)THEN
           dlam=dl(i,iel); CALL vmflow(stress,dsbar,vmfl)
           CALL fmrmat(vmfl,dsbar,dlam,dee,rmat); caflow=MATMUL(rmat,vmfl)
           bot=DOT_PRODUCT(vmfl,caflow); CALL formaa(vmfl,rmat,daatd)
           dee=rmat-daatd/bot
         END IF; sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
         CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated------------------
         fnew=dsbar-SQRT(d3)*prop(1,etype(iel)); fstiff=fnew; elso=zero
         IF(fnew>=zero)THEN
           CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
           CALL vmflow(stress,dsbar,vmfl); caflow=MATMUL(dee,vmfl)
           bot=DOT_PRODUCT(vmfl,caflow); dlam=fnew/bot; elso=caflow*dlam
           stress=tensor(:,i,iel)+sigma-elso
           CALL invar(stress,sigm,dsbar,lode_theta)
           fnew=dsbar-SQRT(d3)*prop(1,etype(iel))
           iterate_on_fnew: DO
             CALL vmflow(stress,dsbar,vmfl); caflow=MATMUL(dee,vmfl)*dlam
             ress=stress-(tensor(:,i,iel)+sigma-caflow)
             CALL fmacat(vmfl,acat); acat=acat/dsbar
             acatc=MATMUL(dee,acat); qmat=acatc*dlam
             DO k=1,4; qmat(k,k)=qmat(k,k)+one; END DO; CALL invert(qmat)
             vmtemp(1,:)=vmfl; vmtemp=MATMUL(vmtemp,qmat)
             vmflq=vmtemp(1,:); top=DOT_PRODUCT(vmflq,ress)
             vmtemp=MATMUL(vmtemp,dee); vmfla=vmtemp(1,:)
             bot=DOT_PRODUCT(vmfla,vmfl); dslam=(fnew-top)/bot
             qinvr=MATMUL(qmat,ress); qinva=MATMUL(MATMUL(qmat,dee),vmfl)
             dsigma=-qinvr-qinva*dslam; stress=stress+dsigma
             CALL invar(stress,sigm,dsbar,lode_theta)
             fnew=dsbar-SQRT(d3)*prop(1,etype(iel)); dlam=dlam+dslam
             IF(fnew<tol)EXIT
           END DO iterate_on_fnew
           dl(i,iel)=dlam; elso=tensor(:,i,iel)+sigma-stress
           eload=MATMUL(elso,bee); bload=bload+eload*det*weights(i)
           CALL vmflow(stress,dsbar,vmfl)
           CALL fmrmat(vmfl,dsbar,dlam,dee,rmat); caflow=MATMUL(rmat,vmfl)
           bot=DOT_PRODUCT(vmfl,caflow); CALL formaa(vmfl,rmat,daatd)
           dee=rmat-daatd/bot
         END IF
         IF(fstiff<zero)                                              &
           CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
         km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----------------------update the Gauss Point stresses-------------------
         tensor(:,i,iel)=tensor(:,i,iel)+sigma-elso; stress=tensor(:,i,iel)
         eload=MATMUL(stress,bee); dload=dload+eload*det*weights(i)
       END DO gauss_points_2
!-----------------------compute the total bodyloads vector----------------
       bdylds(g)=bdylds(g)+bload; bdylds(0)=zero; ddylds(g)=ddylds(g)+dload
       ddylds(0)=zero; storkm(:,:,iel)=km
       DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
     END DO elements_4; diag_precon(1:neq)=one/diag_precon(1:neq)
     diag_precon(0)=zero; tloads=SUM(bdylds); IF(iters==1)converged=.FALSE.
     IF(iters/=1.AND.tloads<ltol)converged=.TRUE.; totd=totd+loads
     IF(converged.OR.iters==limit)EXIT
   END DO plastic_iters; totd=totd+loads
   WRITE(11,'(I5,2E12.4,I5,F17.2)')                                     &
         iy,ptot,totd(nf(2,no(1))),iters,REAL(cg_tot)/REAL(iters)
   IF(iters==limit)EXIT
 END DO load_increments
 CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(totd,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p66
```

The “mesh-free” version of the constant stiffness method, Program 6.2, was inefficient (at least on a scalar computer) due to the large number of repeated equation solutions. Program 6.5 has shown that a tangent stiffness approach, with consistent return, has reduced the number of equation solutions to 4 per load increment. With significantly fewer equations to be solved, there is less benefit to be gained from factorisation, implying that a “mesh-free” approach using an iterative preconditioned conjugate gradient solver could be appropriate. Program 6.6 should be seen as a merging of Programs 6.2 and 6.5. The data for the problem solved, again the original one of Figure 6.9, are shown in Figure 6.27. Extra information,

```
nxe   nye   cg_tol   cg_limit    fftol     ltol  np_types
8     4     0.0001     100      -1.0e-6  5.0e-5   1

prop(cu,e,v)
100.0  1.0e5  0.3

etype(not needed)

x_coords, y_coords
0.0   1.0   2.0   3.0   4.0   5.5   7.0   9.0  12.0
0.0  -1.25 -2.5  -3.75 -5.0

nr,(k,nf(:,k),i=1,nr)
33
  1 0 1    2 0 1    3 0 1    4 0 1    5 0 1    6 0 1
  7 0 1    8 0 1    9 0 0   14 0 0   23 0 0   28 0 0
 37 0 0   42 0 0   51 0 0   56 0 0   65 0 0   70 0 0
 79 0 0   84 0 0   93 0 0   98 0 0  107 0 0  112 0 0
113 0 0  114 0 0  115 0 0  116 0 0  117 0 0  118 0 0
119 0 0  120 0 0  121 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
5
1   0.0  -0.166667     10  0.0  -0.666667     15  0.0  -0.333333
24  0.0  -0.666667     29  0.0  -0.166667

tol   limit
0.001  50

incs,(qinc(i),i=1,incs)
10
200.0 100.0  50.0  50.0  50.0  30.0  20.0  10.0   5.0   5.0
```

Figure 6.27 Data for Program 6.6 example

```
 There are    184  equations

   step   load        disp      iters    cg iters/plastic iter
     1  0.2000E+03 -0.6593E-02    1           46.00
     2  0.3000E+03 -0.1154E-01    4           51.50
     3  0.3500E+03 -0.1614E-01    4           56.00
     4  0.4000E+03 -0.2284E-01    4           63.00
     5  0.4500E+03 -0.3277E-01    4           69.50
     6  0.4800E+03 -0.4183E-01    4           78.75
     7  0.5000E+03 -0.5031E-01    4           78.25
     8  0.5100E+03 -0.5614E-01    4           88.50
     9  0.5150E+03 -0.6062E-01    4           84.75
    10  0.5200E+03 -0.2117E+00   17           99.59
```

Figure 6.28 Results from Program 6.6 example

as compared with the data for Program 6.5, is limited to the conjugate gradient tolerance and iteration limit, set respectively to `cg_tol=0.0001` and `cg_limit=100`.

The output is listed as Figure 6.28 which can be compared with Figure 6.26. Between 46 and 100 conjugate gradient iterations per plastic iteration were required, but the program ran faster than Program 6.1 for the solution of this problem, even in scalar mode. In parallel implementations, there is a trade-off between constant stiffness and tangent stiffness methods because for the latter, all yielded elements are different, although their geometries and elastic properties may be identical.

6.11 The geotechnical processes of embanking and excavation

6.11.1 Embanking

One of the main features of analyses of geotechnical problems is the need to model construction processes. Gravity is one of the main agencies causing deformations and it is common to employ “gravity turn-on” as the loading mechanism. In embankments for example, the final geometry of a weightless slope can be modelled by a finite element mesh, which is then subjected to gravity loading, often in a single increment. To obtain the Factor of Safety, the soil’s strength parameters can be reduced sequentially to failure (see e.g. Program 6.3).

Although it has been known for a long time (Smith and Hobbs, 1974) that this method can capture some of the realities of a construction process which takes place piece by piece (e.g. in layers), it is more realistic to be able to build up a mesh in stages modelling the influence of gravity at each stage.

Program 6.7 Plane strain construction of an elastic–plastic (Mohr–Coulomb) embankment in layers on a foundation using 8-node quadrilaterals. Viscoplastic strain method.

```
PROGRAM p67
!-------------------------------------------------------------------------
! Program 6.7 Plane strain construction of an elastic-plastic
!             (Mohr-Coulomb) embankment in layers on a foundation using
!             8-node quadrilaterals. Viscoplastic strain method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::enxe,enye,fnxe,fnye,i,iel,ii,incs,iters,itype,iy,k,lifts,limit, &
   lnn,ndim=2,ndof=16,nels,neq,newele,nip=4,nn,nod=8,nodof=2,nr,nst=4,    &
   oldele,oldnn
 REAL(iwp)::c,c_e,c_f,det,dq1,dq2,dq3,dsbar,dt,d4=4.0_iwp,d180=180.0_iwp, &
   e,e_e,e_f,f,gamma,gama_e,gama_f,k0,lode_theta,one=1.0_iwp,phi,phi_e,   &
   phi_f,pi,psi,psi_e,psi_f,sigm,snph,tol,two=2.0_iwp,v,v_e,v_f,          &
   zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),lnf(:,:),&
   nf(:,:),num(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   der(:,:),deriv(:,:),devp(:),edepth(:),eld(:),eload(:),eps(:),erate(:), &
   evp(:),evpt(:,:,:),ewidth(:),fdepth(:),flow(:,:),fun(:),fwidth(:),     &
   gc(:),gravlo(:),g_coord(:,:),jac(:,:),km(:,:),kv(:),loads(:),m1(:,:),  &
   m2(:,:),m3(:,:),oldis(:),points(:,:),sigma(:),stress(:),tensor(:,:,:), &
   totd(:),weights(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)fnxe,fnye,nn,incs,limit,tol,lifts,enxe,enye,itype,k0,e_f,v_f,  &
   c_f,phi_f,psi_f,gama_f,e_e,v_e,c_e,phi_e,psi_e,gama_e
!-----------------------calculate the total number of elements------------
 k=0; DO i=1,enye-1; k=i+k; END DO; nels=fnxe*fnye+(enxe*enye-k)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   edepth(enye+1),num(nod),dee(nst,nst),evpt(nst,nip,nels),ewidth(enxe+1),&
   coord(nod,ndim),fun(nod),etype(nels),g_g(ndof,nels),jac(ndim,ndim),    &
   der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),           &
   km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof),eload(ndof),   &
   erate(nst),evp(nst),devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),         &
   m3(nst,nst),flow(nst,nst),stress(nst),fwidth(fnxe+1),fdepth(fnye+1),   &
   gc(ndim),tensor(nst,nip,nels))
 READ(10,*)fwidth,fdepth,ewidth,edepth
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 WRITE(11,'(A,I5)')" The final number of elements is:",nels
 WRITE(11,'(A,I5)')" The final number of freedoms is:",neq
!-----------------------set the element type-----------------------------
 etype(1:fnxe*fnye)=1; etype(fnxe*fnye+1:nels)=2
!-----------set up the global node numbers and element nodal coordinates--
 CALL fmglem(fnxe,fnye,enxe,g_num,lifts)
 CALL fmcoem(g_num,g_coord,fwidth,fdepth,ewidth,edepth,                   &
   enxe,lifts,fnxe,fnye,itype)
 ALLOCATE(totd(0:neq)); tensor=zero; totd=zero; pi=ACOS(-one)
!-----------------------loop the elements to find the global g------------
 elements_1: DO iel=1,nels
   num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
 END DO elements_1; CALL sample(element,points,weights)
!-----------------------construct another lift----------------------------
 lift_number: DO ii=1,lifts
   WRITE(11,'(/A,I5)')" Lift number",ii
!-----------------------calculate how many elements there are-------------
   IF(ii<=lifts)THEN
     IF(ii==1)THEN; newele=fnxe*fnye; oldele=newele; ELSE
       newele=enxe-(ii-2); oldele=oldele+newele
     END IF
!-----------------------calculate how many nodes there are----------------
     IF(ii==1)THEN; lnn=(fnxe*2+1)*(fnye+1)+(fnxe+1)*fnye; oldnn=lnn
     END IF
     IF(ii>1)THEN; lnn=oldnn+(enxe-(ii-2))*2+1+(enxe-(ii-2)+1); oldnn=lnn
     END IF; ALLOCATE(lnf(nodof,lnn)); lnf=nf(:,1:lnn)
!-----------------------recalculate the number of freedoms neq------------
     neq=MAXVAL(lnf); ALLOCATE(kdiag(neq)); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
     elements_2: DO iel=1,oldele; g=g_g(:,iel); CALL fkdiag(kdiag,g)
     END DO elements_2
     DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
     WRITE(11,'(3(A,I5))')" There are",neq," freedoms"
     WRITE(11,'(3(A,I5))')" There are",oldele," elements after",          &
       newele," were added"
   END IF
   ALLOCATE(kv(kdiag(neq)),loads(0:neq),bdylds(0:neq),oldis(0:neq),       &
     gravlo(0:neq)); gravlo=zero; loads=zero; kv=zero
!-----------------------element stiffness integration and assembly--------
   elements_3: DO iel=1,oldele
     IF(etype(iel)==1)THEN; gamma=gama_f; e=e_f; v=v_f
     ELSE; gamma=gama_e; e=e_e; v=v_e
     END IF
     IF(iel<=(oldele-newele))gamma=zero; num=g_num(:,iel)
     coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
     CALL deemat(dee,e,v); eld=zero
     gauss_pts_1: DO i=1,nip
       CALL shape_fun(fun,points,i); gc=MATMUL(fun,coord)
!-----------------------initial stress in foundation----------------------
      IF(ii==1)THEN; tensor(2,i,iel)=-one*(fdepth(fnye+1)-gc(2))*gamma
```

```
        tensor(1,i,iel)=k0*tensor(2,i,iel)
        tensor(4,i,iel)=tensor(1,i,iel); tensor(3,i,iel)=zero
      END IF
      CALL shape_der(der,points,i)
      jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
      deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
      km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
      DO k=2,ndof,2; eld(k)=eld(k)+fun(k/2)*det*weights(i); END DO
    END DO gauss_pts_1
    CALL fsparv(kv,km,g,kdiag)
    IF(ii<=lifts)gravlo(g)=gravlo(g)-eld*gamma; gravlo(0)=zero
  END DO elements_3
!-----------------------factorise equations--and-factor gravlo by incs----
  CALL sparin(kv,kdiag); gravlo=gravlo/incs
!-----------------------apply gravity loads incrementally-----------------
  load_incs: DO iy=1,incs
    iters=0; oldis=zero; bdylds=zero; evpt(:,:,1:oldele)=zero
!-----------------------iteration loop-----------------------------------
    its: DO
      iters=iters+1
      WRITE(*,'(A,I3,A,I3,A,I4)')"  lift",ii,"  increment",iy,           &
        "  iteration",iters
      loads=zero; loads=gravlo+bdylds; CALL spabac(kv,loads,kdiag)
!-----------------------check convergence--------------------------------
      CALL checon(loads,oldis,tol,converged)
      IF(iters==1)converged=.FALSE.
      IF(converged.OR.iters==limit)bdylds=zero
!-----------------------go round the Gauss Points------------------------
      elements_4: DO iel=1,oldele
        IF(etype(iel)==1)THEN; phi=phi_f; c=c_f; e=e_f; v=v_f; psi=psi_f
        ELSE; phi=phi_e; c=c_e; e=e_e; v=v_e; psi=psi_e
        END IF; snph=SIN(phi*pi/d180)
        dt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2))
        CALL deemat(dee,e,v); bload=zero; num=g_num(:,iel)
        coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
        gauss_pts_2: DO i=1,nip
          CALL shape_der(der,points,i)
          jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
          deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
          eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
          sigma=MATMUL(dee,eps)
          IF(ii==1)THEN; stress=tensor(:,i,iel)
          ELSE; stress=tensor(:,i,iel)+sigma
          END IF; CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
          CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
          IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
            IF(f>=zero)THEN
              CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
              CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
              erate=MATMUL(flow,stress); evp=erate*dt
              evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
            END IF
          END IF
          IF(f>=zero)THEN; eload=MATMUL(devp,bee)
            bload=bload+eload*det*weights(i)
          END IF
!-----------------------if appropriate update the Gauss point stresses----
```

```
           IF(converged.OR.iters==limit)THEN
             IF(ii/=1)tensor(:,i,iel)=stress
           END IF
         END DO gauss_pts_2
!-----------------------compute the total bodyloads vector----------------
         bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
       END DO elements_4; IF(converged.OR.iters==limit)EXIT
     END DO its; IF(ii/=1)totd(:neq)=totd(:neq)+loads(:neq)
     WRITE(11,'(2(A,I5),A)')" Increment",iy," took ",iters,            &
       " iterations to converge"
     IF(iy==incs.OR.iters==limit)WRITE(11,'(A,E12.4)')                 &
       " Max displacement is",MAXVAL(ABS(loads))
     IF(iters==limit)THEN
       CALL dismsh(loads,lnf,0.05_iwp,g_coord,g_num,13)
       CALL vecmsh(loads,lnf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
       STOP
     END IF
   END DO load_incs; DEALLOCATE(lnf,kdiag,kv,loads,bdylds,oldis,gravlo)
 END DO lift_number
STOP
END PROGRAM p67
```

New scalar integers:

enxe	number of x-elements in embankment
enye	number of y-elements in embankment
fnxe	number of x-elements in foundation
fnye	number of y-elements in foundation
ii	counts the lifts
itype	type of degeneration of quadrilateral to triangle
lifts	number of lifts
lnn	keeps running total of number of nodes
newele	number of new elements at each lift
oldele	keeps running total of number of elements
oldnn	number of nodes from previous lift

New scalar reals:

c_e	embankment cohesion
c_f	foundation cohesion
e_e	Young's modulus in embankment
e_f	Young's modulus in foundation
gama_e	unit weight of embankment
gama_f	unit weight of foundation
phi_e	friction angle of embankment
phi_f	friction angle of foundation
psi_e	dilation angle of embankment
psi_f	dilation angle of foundation
v_e	Poisson's ratio of embankment
v_f	Poisson's ratio of foundation

New dynamic integer arrays:

lnf	updated nodal freedom array

New dynamic real arrays:

`edepth`	y-coordinates of top of each embankment lift
`ewidth`	x-coordinates of base of embankment
`fdepth`	y-coordinates of foundation depth layout
`fwidth`	x-coordinates of foundation depth layout

Program 6.7 analyses the stability of a slope, built up in layers on a foundation. Figure 6.29 shows a typical staged construction of an embankment (plane strain) on a rectangular foundation block of soil. The embankment is assumed to be raised in a series of “lifts”, the first of which merely stresses the foundation block gravitationally under “at rest” conditions. The “soil” is assumed to be an elastic–plastic Mohr–Coulomb material and a viscoplastic strain algorithm is used, so the program can be considered to be a development of Program 6.3. Figure 6.30 shows the final mesh in more detail with the input data for the problem.

The basic mesh information is read in the data with `fnxe, fnye, fwidth` and `fdepth` for the foundation, and `enxe, enye, ewidth` and `edepth` for the embankment. Subroutines `fmglem` and `fmcoem` set up the global node numbers and element

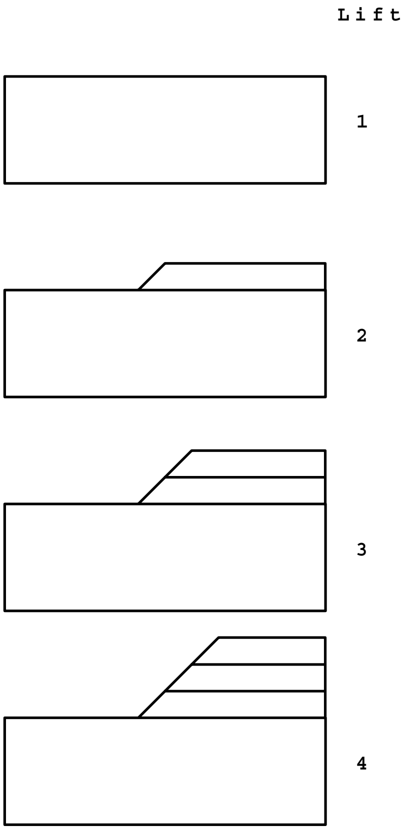
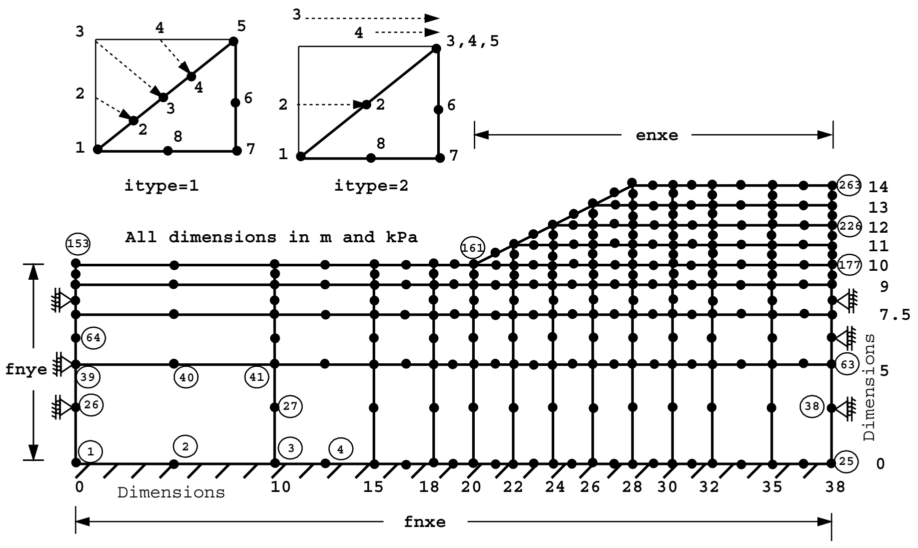


Figure 6.29 Staged construction of an embankment



```
fnxe fnye   nn   incs limit   tol
12     4    263    3    500   0.0001

lifts enxe enye itype  k0
5       8    4     1   1.0

e_f     v_f    c_f    phi_f  psi_f gama_f
1.0e5   0.49   14.0    0.0    0.0   20.0

e_e     v_e    c_e    phi_e  psi_e gama_e
1.0e5   0.49   14.0    0.0    0.0   20.0

fwidth, fdepth, ewidth, edepth
 0.0  10.0  15.0  18.0  20.0  22.0  24.0  26.0  28.0  30.0  32.0  35.0  38.0
 0.0   5.0   7.5   9.0  10.0
20.0  22.0  24.0  26.0  28.0  30.0  32.0  35.0  38.0
10.0  11.0  12.0  13.0  14.0

nr,(k,nf(:,k),i=1,nr)
49
  1 0 0    2 0 0    3 0 0    4 0 0    5 0 0    6 0 0   7 0 0   8 0 0
  9 0 0   10 0 0   11 0 0   12 0 0   13 0 0   14 0 0  15 0 0  16 0 0
 17 0 0   18 0 0   19 0 0   20 0 0   21 0 0   22 0 0  23 0 0  24 0 0
 25 0 0   26 0 1   38 0 1   39 0 1   63 0 1   64 0 1  76 0 1  77 0 1
101 0 1  102 0 1  114 0 1  115 0 1  139 0 1  140 0 1 152 0 1 153 0 1
177 0 1  186 0 1  203 0 1  211 0 1  226 0 1  233 0 1 246 0 1 252 0 1
263 0 1
```

Figure 6.30 Mesh and data for Program 6.7 example

nodal coordinates g_num and g_coord respectively, in this case customised for a 2:1 slope inclined at 26.57° to the horizontal. Alternatives, controlled by itype as shown in the figure, allow quadrilaterals to be degenerated into triangles (on the face of the embankment slope) by two different methods.

The problem consists of two materials, the foundation, with properties given by e_f, v_f, c_f, phi_f, psi_f and gama_f and the embankment, with properties given by e_e, v_e, c_e, phi_e, psi_e and gama_e.

Because the mesh is updated at every "lift" there is a need for a "local" node freedom array lnf which is found from the final nf at every stage. lnf is ALLOCATEd and

DEALLOCATEd at each lift. Then, for each lift the geometry and connectivity can be calculated and hence the number of equations, neq and stiffness matrix diagonal locator kdiag, operating at that stage of the construction process. The stiffness matrix and load vector sizes can then be set by an ALLOCATE statement and the viscoplastic algorithm initiated. The program has the option of applying the gravity loads associated with each lift in incs increments (3 in this case).

The results are shown in Figure 6.31 and plotted in Figure 6.32, showing the progress of toe displacement as the embankment is raised. For the case shown, in which the foundation and embankment are both undrained clays, with $c_u = 14$ kN/m^2, the embankment is seen to fail when its height reaches approximately 4 m. This is exactly what would be predicted by Taylor's (1937) charts, which gives, for a 2:1 slope with a depth ratio of $D = 3.5$, a stability number equal to 0.18, implying a factor of safety very close to unity. The displacement vectors corresponding to the unconverged solution when the embankment height reached 4 m are shown in Figure 6.33.

```
The final number of elements is:   74
The final number of freedoms is:  452

Lift number    1
There are  288 freedoms
There are   48 elements after   48 were added
Increment    1 took     2 iterations to converge
Increment    2 took     2 iterations to converge
Increment    3 took     2 iterations to converge
Max displacement is  0.1948E-03

Lift number    2
There are  338 freedoms
There are   56 elements after    8 were added
Increment    1 took     2 iterations to converge
Increment    2 took     2 iterations to converge
Increment    3 took     2 iterations to converge
Max displacement is  0.2624E-03

Lift number    3
There are  382 freedoms
There are   63 elements after    7 were added
Increment    1 took     2 iterations to converge
Increment    2 took     2 iterations to converge
Increment    3 took     2 iterations to converge
Max displacement is  0.2806E-03

Lift number    4
There are  420 freedoms
There are   69 elements after    6 were added
Increment    1 took     2 iterations to converge
Increment    2 took     5 iterations to converge
Increment    3 took     7 iterations to converge
Max displacement is  0.3311E-03

Lift number    5
There are  452 freedoms
There are   74 elements after    5 were added
Increment    1 took    15 iterations to converge
Increment    2 took    45 iterations to converge
Increment    3 took   500 iterations to converge
Max displacement is  0.1138E+00
```

Figure 6.31 Results from Program 6.7 example

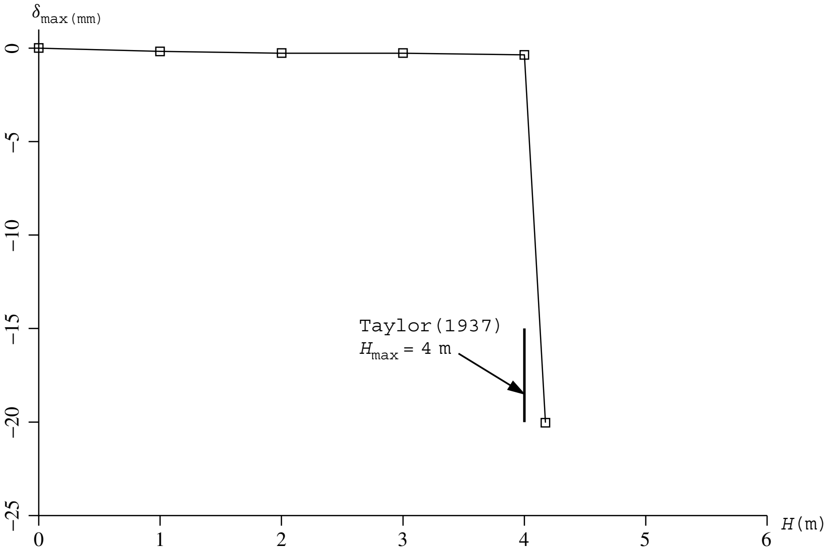


Figure 6.32 Embankment height versus maximum nodal displacement from Program 6.7 example

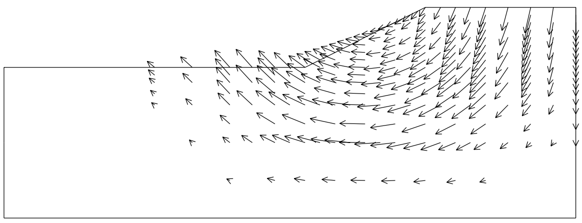


Figure 6.33 Displacement vectors corresponding to the unconverged solution when the embankment height reached 4 m

6.11.2 Excavation

The second important geotechnical construction process involving change of geometry occurs when material is removed from the ground, either in open excavations ("cuts") or in enclosed tunnels. The ground is stressed prior to removal of part of it and this starting stress state may be difficult to infer from the known history.

The aim in an analysis is that, when a portion of material is excavated, and forces are applied along the excavated surface, the remaining material should experience the correct stress relief so that the new "free surface" is indeed stress-free.

Suppose body A is to be removed from body B as shown in Figure 6.34. The stresses to begin with are $\{\boldsymbol{\sigma}_{A0}\}$ and $\{\boldsymbol{\sigma}_{B0}\}$ respectively. Any external loads are taken into consideration in forming these stresses prior to the removal of A. Since both bodies are in equilibrium, forces $\{\mathbf{F}_{AB}\}$ must be applied to body B due to body A to maintain $\{\boldsymbol{\sigma}_{B0}\}$ and, similarly,

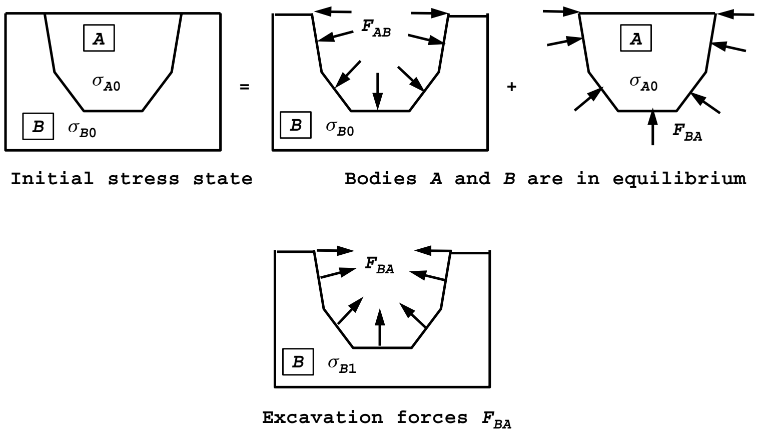


Figure 6.34 Formulation of excavation forces

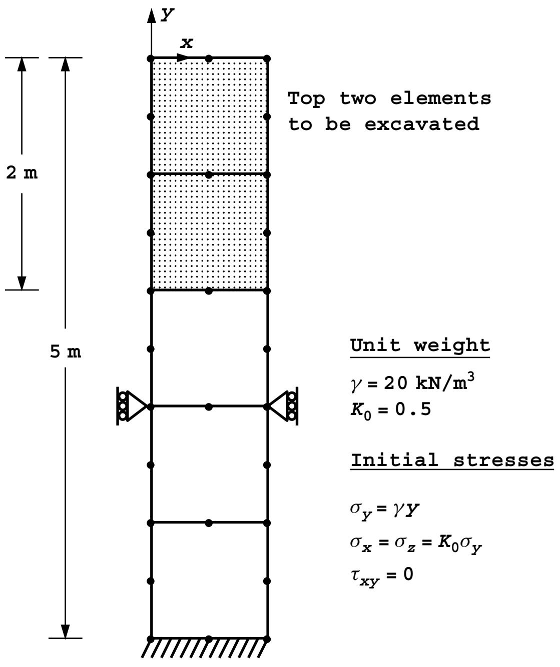


Figure 6.35 Column of elements before excavation

{$\mathbf{F}_{\mathrm{BA}}$} must act on body A. Forces {$\mathbf{F}_{\mathrm{AB}}$} and {$\mathbf{F}_{\mathrm{BA}}$} are equal in magnitude and opposite in sign. In general, therefore, the excavation forces acting on a boundary depend on the stress state in the excavated material and on the self-weight of that material. It can be shown that,

$$\{\mathbf{F}_{\mathrm{BA}}\} = \int_{V_{\mathrm{A}}} [\mathbf{B}]^{\mathrm{T}} \{\boldsymbol{\sigma}_{\mathrm{A0}}\} \, \mathrm{d}V_{\mathrm{A}} + \gamma \int_{V_{\mathrm{A}}} [\mathbf{N}]^{\mathrm{T}} \mathrm{d}V_{\mathrm{A}} \qquad (6.75)$$

where [**B**] is the strain-displacement matrix, V_A the excavated volume, [**N**] the element shape functions and γ the soil unit weight.

Single-stage and multi-stage excavations give the same results, and in a one-dimensional situation a stress-free excavated surface results. Figure 6.35 shows a column of five 8-node elements at rest under self-weight stresses from which the top two elements are to be excavated. Figure 6.36 shows the excavation force terms computed using 2 × 2 Gauss points from (6.75), and the resulting forces that need to be applied to the free surface.

When the situation is two- or three-dimensional, equation (6.75) applies again, but in the corners of excavations, a rather complex stress concentration exists. This means that the finite element results will be mesh-dependent (Smith and Ho, 1992).

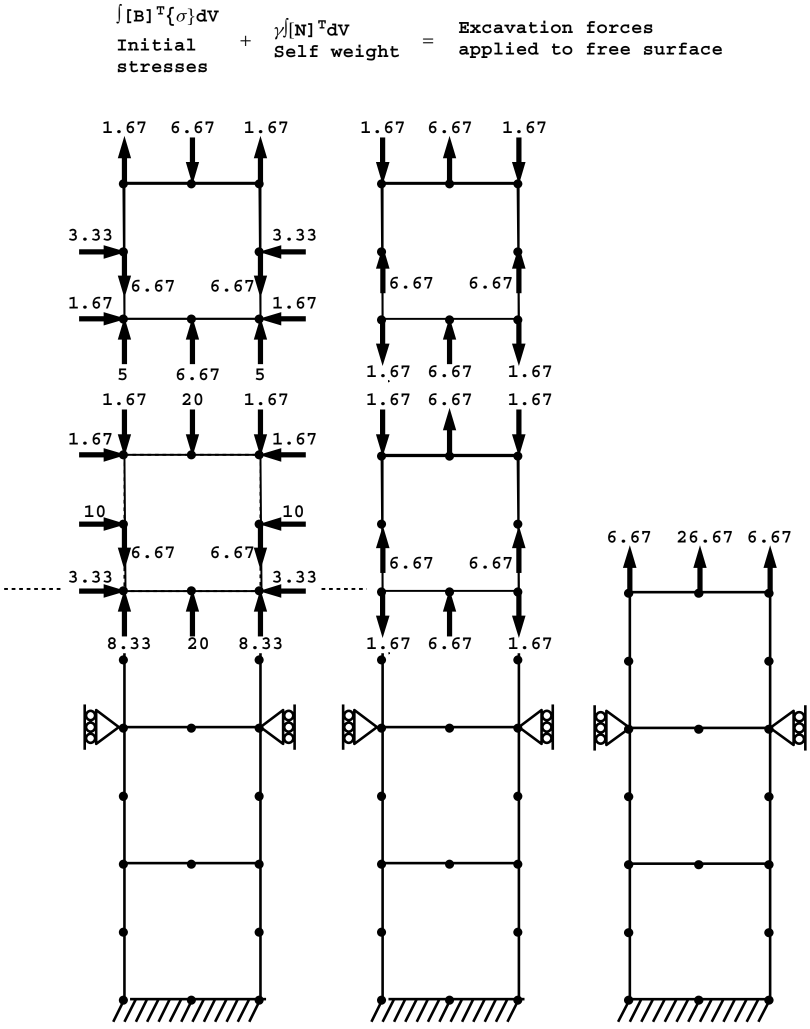


Figure 6.36 Development of excavation force terms

A program illustrating the analysis of excavation processes is listed as Program 6.8. As with the previous program, it can be derived from Program 6.3.

Program 6.8 Plane strain construction of an elastic–plastic (Mohr–Coulomb) excavation in layers using 8-node quadrilaterals. Viscoplastic strain method.

```
PROGRAM p68
!-------------------------------------------------------------------------
! Program 6.8 Plane strain construction of an elastic-plastic
!             (Mohr-Coulomb) excavation in layers using 8-node
!             quadrilaterals. Viscoplastic strain method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,ii,incs,iters,iy,k,layers,limit,ndim=2,ndof=16,nels,neq,  &
   nip=4,nn,nod=8,nodof=2,noexe,nouts,nprops=7,np_types,nr,nst=4,ntote
 REAL(iwp)::c,ddt,det,dq1,dq2,dq3,dsbar,dt,d4=4.0_iwp,d180=180.0_iwp,e,f, &
   gamma,lode_theta,one=1.0_iwp,phi,pi,psi,sigm,snph,start_dt=1.e15_iwp,  &
   tol,two=2.0_iwp,v,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!----------------------dynamic arrays----------------------------------
 INTEGER,ALLOCATABLE::etype(:),exele(:),g(:),g_num(:,:),kdiag(:),lnf(:,:),&
   nf(:,:),no(:),num(:),solid(:),totex(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   der(:,:),deriv(:,:),devp(:),eld(:),eload(:),eps(:),erate(:),evp(:),    &
   evpt(:,:,:),exc_loads(:),flow(:,:),fun(:),gc(:),g_coord(:,:),jac(:,:), &
   km(:,:),kv(:),loads(:),m1(:,:),m2(:,:),m3(:,:),oldis(:),points(:,:),   &
   prop(:,:),stress(:),tensor(:,:,:),tot_d(:,:),weights(:)
!----------------------input and initialisation------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,nn,np_types; ALLOCATE(prop(nprops,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod),     &
   solid(nels),jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),              &
   g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof),eps(nst),        &
   totex(nels),bload(ndof),eload(ndof),erate(nst),evp(nst),devp(nst),     &
   g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),stress(nst), &
   tot_d(nodof,nn),gc(ndim),tensor(nst,nip,nels),lnf(nodof,nn))
!----------------------read geometry and connectivity-------------------
 READ(10,*)g_coord,g_num
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr)
!---------------lnf is the current nf at each stage of excavation---------
 lnf=nf; CALL formnf(lnf); neq=MAXVAL(lnf)
 WRITE(11,'(A,I5)')" The initial number of elements is:",nels
 WRITE(11,'(A,I5)')" The initial number of freedoms is:",neq
!--------set up the global node numbers and global nodal coordinates------
!----------------------loop the elements to set starting stresses--------
 CALL sample(element,points,weights)
 elements_0: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   gamma=prop(4,etype(iel))
   gauss_pts_0: DO i=1,nip
     CALL shape_fun(fun,points,i); gc=MATMUL(fun,coord)
     tensor(2,i,iel)=gc(2)*gamma
     tensor(1,i,iel)=prop(7,etype(iel))*tensor(2,i,iel)
     tensor(4,i,iel)=tensor(1,i,iel); tensor(3,i,iel)=zero
   END DO gauss_pts_0
 END DO elements_0; tot_d=zero; ntote=0; solid=1; totex=0; pi=ACOS(-one)
!-----------------------excavate a layer---------------------------------
 READ(10,*)nouts; ALLOCATE(no(nouts)); READ(10,*)no,tol,limit,incs,layers
 layer_number: DO ii=1,layers
   WRITE(11,'(/A,I5)')" Excavation number",ii
!-----------------------read elements to be removed----------------------
   READ(10,*)noexe; ALLOCATE(exele(noexe)); READ(10,*)exele; solid(exele)=0
   CALL exc_nods(noexe,exele,g_num,totex,ntote,nf); lnf=nf
   CALL formnf(lnf); neq=MAXVAL(lnf)
   ALLOCATE(kdiag(neq),exc_loads(0:neq),bdylds(0:neq),oldis(0:neq),       &
     loads(0:neq)); WRITE(11,'(3(A,I5))')" There are",neq," freedoms"
   WRITE(11,'(3(A,I5))')" There are",nels-ntote," elements after",        &
     noexe," were removed"; kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
   elements_1: DO iel=1,nels
     num=g_num(:,iel); CALL num_to_g(num,lnf,g); CALL fkdiag(kdiag,g)
   END DO elements_1
   DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
   ALLOCATE(kv(kdiag(neq))); exc_loads=zero
!-----------------------calculate excavation load ------------------------
   elements_2: DO iel=1,noexe
     k=exele(iel); gamma=prop(4,etype(k)); bload=zero; eld=zero
     num=g_num(:,k); CALL num_to_g(num,lnf,g)
     coord=TRANSPOSE(g_coord(:,num))
     gauss_pts_2: DO i=1,nip
       CALL shape_fun(fun,points,i); stress=tensor(:,i,k)
       CALL bee8(bee,coord,points(i,1),points(i,2),det)
       eload=MATMUL(stress,bee); bload=bload+eload*det*weights(i)
       eld(nodof:ndof:nodof)=eld(nodof:ndof:nodof)+fun(:)*det*weights(i)
     END DO gauss_pts_2; exc_loads(g)=exc_loads(g)+eld*gamma+bload
   END DO elements_2; exc_loads(0)=zero
!-----------------------element stiffness integration and assembly--------
   kv=zero; dt=start_dt
   elements_3: DO iel=1,nels
     IF(solid(iel)==0)THEN; e=zero; ELSE
       phi=prop(1,etype(iel)); e=prop(5,etype(iel)); v=prop(6,etype(iel))
       snph=SIN(phi*pi/d180)
       ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph*snph))
       IF(ddt<dt)dt=ddt
     END IF
     km=zero; eld=zero; CALL deemat(dee,e,v); num=g_num(:,iel)
     CALL num_to_g(num,lnf,g); coord=TRANSPOSE(g_coord(:,num))
     gauss_pts_3: DO i=1,nip
       CALL bee8(bee,coord,points(i,1),points(i,2),det)
       km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
     END DO gauss_pts_3; CALL fsparv(kv,km,g,kdiag)
   END DO elements_3
!-----------------------factorise sand factor excavation load by incs-----
   CALL sparin(kv,kdiag); exc_loads=exc_loads/incs
!-----------------------apply excavation loads incrementally--------------
   load_incs: DO iy=1,incs
     iters=0; oldis=zero; bdylds=zero; evpt=zero
!-----------------------iteration loop -----------------------------------
     its: DO
       iters=iters+1
       WRITE(*,'(A,I3,A,I3,A,I4)')"  excavation",ii,"  increment",iy,     &
         "  iteration",iters
       loads=exc_loads+bdylds; CALL spabac(kv,loads,kdiag)
!-----------------------check convergence---------------------------------
       CALL checon(loads,oldis,tol,converged)
       IF(iters==1)converged=.FALSE.
       IF(converged.OR.iters==limit)THEN; bdylds=zero
         DO k=1,nn; DO i=1,nodof
            IF(lnf(i,k)/=0)tot_d(i,k)=tot_d(i,k)+loads(lnf(i,k))
         END DO; END DO
       END IF
!-----------------------go round the Gauss Points-------------------------
       elements_4: DO iel=1,nels
         phi=prop(1,etype(iel)); c=prop(2,etype(iel))
         psi=prop(3,etype(iel)); e=prop(5,etype(iel))
         v=prop(6,etype(iel)); IF(solid(iel)==0)e=zero; bload=zero
         CALL deemat(dee,e,v); num=g_num(:,iel); CALL num_to_g(num,lnf,g)
         coord=TRANSPOSE(g_coord(:,num)); eld=loads(g)
         gauss_pts_4: DO i=1,nip
           CALL bee8(bee,coord,points(i,1),points(i,2),det)
           eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
           stress=tensor(:,i,iel)+MATMUL(dee,eps)
!-----------------------air element stresses are zero---------------------
           IF(solid(iel)==0)stress=zero
           CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
           CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
           IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
             IF(f>=zero)THEN; CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
               CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
               erate=MATMUL(flow,stress); evp=erate*dt
               evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
             END IF
           END IF
           IF(f>=zero.OR.(converged.OR.iters==limit))THEN
             eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
           END IF
!-----------------------if appropriate update the Gauss point stresses----
           IF(converged.OR.iters==limit)tensor(:,i,iel)=stress
         END DO gauss_pts_4
!-----------------------compute the total bodyloads vector ---------------
         bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
       END DO elements_4; IF(converged.OR.iters==limit)EXIT
     END DO its
     WRITE(11,'(A,I3,A,I5,A)')" Increment",iy," took",iters,              &
                             " iterations to converge"
     IF(iy==incs.OR.iters==limit)THEN
       WRITE(11,'(A)') "  Node   x-disp      y-disp"
       DO i=1,nouts; WRITE(11,'(I5,2E12.4)')no(i),tot_d(:,no(i)); END DO
       EXIT
     END IF
   END DO load_incs; IF(ii==layers.OR.iters==limit)EXIT
   DEALLOCATE(kdiag,kv,exc_loads,bdylds,oldis,loads,exele)
 END DO layer_number
 loads(lnf(1,:))=tot_d(1,:); loads(lnf(2,:))=tot_d(2,:)
 g_num(:,totex(:ntote))=0; CALL mesh(g_coord,g_num,12)
 CALL dismsh(loads,lnf,0.1_iwp,g_coord,g_num,13)
 CALL vecmsh(loads,lnf,0.1_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p68
```

New scalar integers:

`layers`	number of excavation steps
`noexe`	number of elements to be removed at each step
`nouts`	number of nodes at which output is required
`ntote`	holds running total of number of excavated elements

New dynamic integer arrays:

`exele`	element numbers of removed elements
`solid`	identifies "air" elements (= 0 for "air", = 1 for solid)
`totex`	holds element numbers for all removed elements

New dynamic real arrays:

`exc_loads`	excavation loads
`tot_d`	holds running total of nodal displacements

Figure 6.37 shows a square block of "soil" from which a vertical cut is to be excavated. The figure indicates five possible sequences that would lead to the same final excavation geometry. Figure 6.38 shows the mesh and data corresponding to excavation case B in a soil with undrained shear strength $c_u = 9$ kN/m^2.

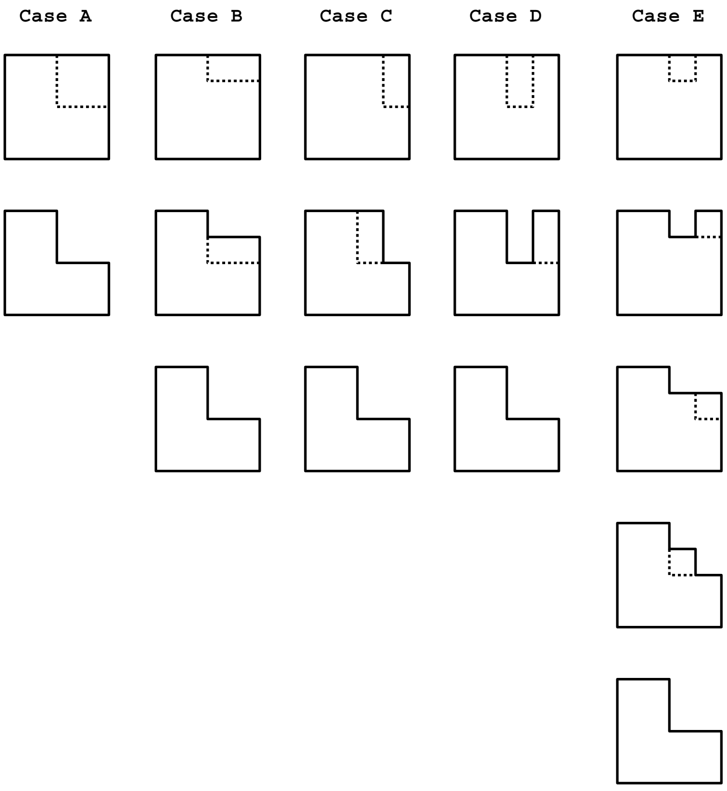
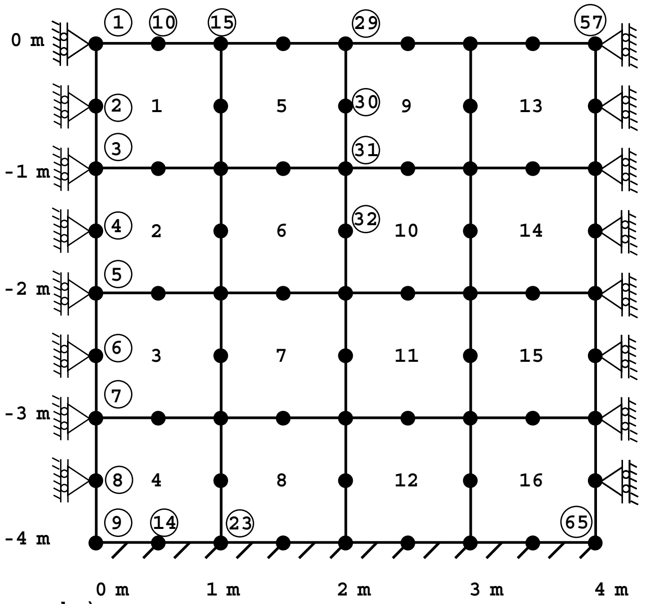


Figure 6.37 Five ways of excavating a vertical cut



```
nels  nn  np_types
16    65    1
```

```
prop(phi,c,psi,gamma,e,v,ko)
0.0  9.0  0.0  20.0  1.0e5  0.49  1.0

etype(not needed)

g_coord
 0.00  0.00    0.00 -0.50    0.00 -1.00    0.00 -1.50    0.00 -2.00
 0.00 -2.50    0.00 -3.00    0.00 -3.50    0.00 -4.00    0.50  0.00
 g_coord data for nodes 11-55 have been omitted here
 3.50 -4.00    4.00  0.00    4.00 -0.50    4.00 -1.00    4.00 -1.50
 4.00 -2.00    4.00 -2.50    4.00 -3.00    4.00 -3.50    4.00 -4.00

g_num
 3    2    1   10   15   16   17   11
 5    4    3   11   17   18   19   12
g_num data for elements 3-14 have been omitted here
49   48   47   54   61   62   63   55
51   50   49   55   63   64   65   56

nr,(k,nf(:,k),i=1,nr)
25
 1 0 1    2 0 1    3 0 1    4 0 1    5 0 1    6 0 1    7 0 1    8 0 1
 9 0 0   14 0 0   23 0 0   28 0 0   37 0 0   42 0 0   51 0 0   56 0 0
57 0 1   58 0 1   59 0 1   60 0 1   61 0 1   62 0 1   63 0 1   64 0 1
65 0 0

nouts,(no(i),i=1,nouts)
4
29 30 31 32

tol    limit  incs  layers
0.0001  250    5      2

noexe,(exele(i),i=1,noexe) (for excavation 1)
2
 9  13

noexe,(exele(i),i=1,noexe) (for excavation 2)
 2
10  14
```

Figure 6.38 Mesh and data for Program 6.8 example

This program allows excavations over general shapes of meshes, so it is the user's responsibility to provide the global coordinates `g_coord` and element numbering `g_num` of the original unexcavated configuration as data. Figure 6.38 shows a truncated data set in the interests of a compact presentation. The same concept of a "local" node freedom array `lnf` as in the previous program is used again. In a preliminary loop, labelled `elements_0` the ground is stressed by its own weight. The soil model allows for 7 properties read as usual into the array `prop`. Note that due to the simplicity of the constitutive model, a very limited range of K_o values could be achieved automatically, so K_o is input as data as the 7th property read into the array `prop`.

Following the coordinates, element numbering, and boundary condition data, the data requires the number of nodes at which output will be required `nouts`, followed by the output node numbers `no`, the plastic tolerance `tol`, the iteration ceiling `limit`, the number of load increments per excavation `incs`, and the number of excavations `layers`.

The final block of data gives the excavation sequence. For each of the excavation steps, the number of elements to be removed `noexe` and the element numbers `exele` must be read. The subroutine `exc_nods` computes the node numbers removed at each excavation step. Excavated "air" elements are given a stiffness of zero ($E = 0$) and the excavated nodes are automatically fully restrained and hence removed from the assembly process.

As in the previous program, for each "layer" the geometry is modified and a new stiffness matrix and load vectors formed. Arrays that change their size from one excavation to the next therefore involve `ALLOCATE` and `DEALLOCATE` statements.

For the case considered, the vertical excavation (case B) is to occur in two steps `layers=2` leading to a cut of depth 2 m. As can be seen from the data, the first excavation removes elements 9 and 13, and the second excavation, removes elements 10 and 14. The output (for case B) shown as Figure 6.39 and plotted in Figure 6.40, indicates that

```
The initial number of elements is:   16
The initial number of freedoms is:   96

Excavation number    1
There are   86 freedoms
There are   14 elements after    2 were removed
Increment  1 took    2 iterations to converge
Increment  2 took    2 iterations to converge
Increment  3 took    2 iterations to converge
Increment  4 took    2 iterations to converge
Increment  5 took    2 iterations to converge
 Node   x-disp      y-disp
  29  0.7635E-05 -0.5876E-04
  30  0.9240E-04 -0.3784E-04
  31  0.8605E-04 -0.1057E-04
  32  0.1102E-03 -0.1652E-05

Excavation number    2
There are   76 freedoms
There are   12 elements after    2 were removed
Increment  1 took    2 iterations to converge
Increment  2 took    2 iterations to converge
Increment  3 took    5 iterations to converge
Increment  4 took   22 iterations to converge
Increment  5 took  250 iterations to converge
 Node   x-disp      y-disp
  29 -0.3073E-03 -0.1641E-02
  30  0.9790E-03 -0.1531E-02
  31  0.2371E-02 -0.1282E-02
  32  0.4044E-02 -0.7897E-03
```

Figure 6.39 Results from Program 6.8 example

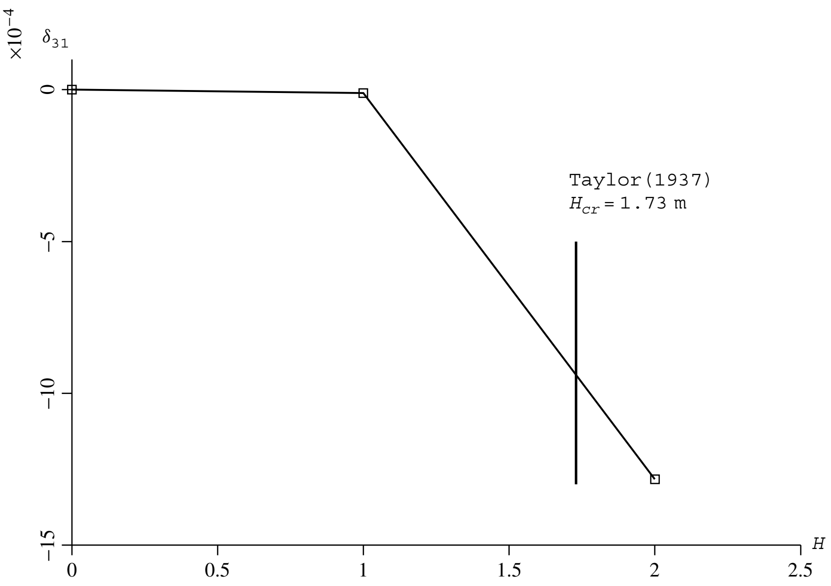


Figure 6.40 Excavation height versus y-displacement at node 31 from Program 6.8 example

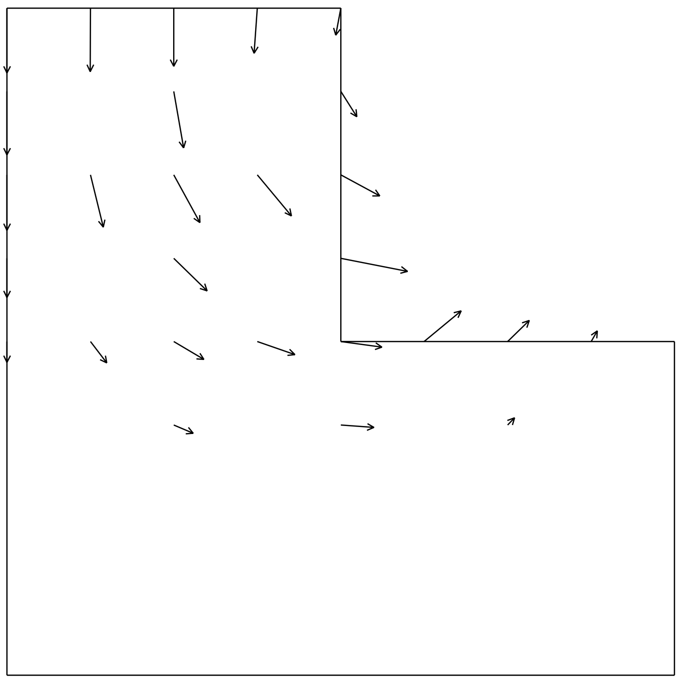


Figure 6.41 Displacement vectors corresponding to the unconverged solution when the excavation height reached 2 m

the displacement at node 31 (on the excavation face) increased very significantly after the second excavation. For a vertical cut consisting of undrained clay with a strength of 9 kN/m^2, Taylor (1937) predicts a critical height of approximately 1.73 m which is well within the range of the second excavation. Figure 6.41 gives the corresponding nodal displacement vectors.

6.12 Undrained analysis

Little mention has been made so far of the role of the dilation angle ψ on the calculation of collapse loads in Mohr–Coulomb materials. The reason is that the dilation angle governs volumetric strains during plastic yield and will have little influence on collapse loads in "unconfined" problems. The examples considered so far in this chapter have been relatively unconfined (e.g. slope stability, earth pressures).

"Undrained" soils are two-phase particulate materials in which the voids between the particles are full of water. In addition, the permeability of the material may be sufficiently low or the loads applied so quickly that pore water pressures that are generated have no time to dissipate during the time scale of the analysis.

In the case of undrained clays that have soft soil skeletons, the shear strength appears to be constant and given by an undrained "cohesion" c_u and $\phi_u = 0$. In such materials, the von Mises or Tresca failure criterion can be successfully applied, as was demonstrated in Program 6.1.

In the case of saturated soils with hard skeletons, such as dense quartz sand, shear stresses will tend to cause dilation which will be resisted by tensile water pressures in the voids of the soil. In turn, the effective stresses between particles will rise and, in a frictional material, the shear stresses necessary to cause failure will also rise. Thus, a dense sand, far from exhibiting a constant shear strength when sheared undrained, would have infinite strength provided the pore fluid could sustain infinite suction and the grains did not crush. In reality, a finite shear strength is recorded due to either grain crushing or pore fluid cavitation.

To perform analyses of this type it is necessary to separate stresses into pore water pressures (isotropic) and effective interparticle stresses (isotropic + shear). Such a treatment has already been described in Section 2.18 in terms of time dependent "consolidation" properties of two-phase materials (Biot's poro-elastic theory) and programs to deal with this will be found in Chapter 9. However, the undrained problem pertaining at the beginning of the Biot process is so important in soil mechanics that it merits special treatment.

Naylor (1974) has described a method of separating the stresses into pore pressures and effective stresses. The method uses as its basis the concept of effective stress in matrix notation; thus

$$\{\boldsymbol{\sigma}\} = \{\boldsymbol{\sigma}'\} + \{\mathbf{u}\} \tag{6.76}$$

where $\{\boldsymbol{\sigma}\}$ is the total stress, $\{\boldsymbol{\sigma}'\}$ the effective stress, and $\{\mathbf{u}\}$ the pore pressure.

The elastic stress–strain relationships can be written as,

$$\{\boldsymbol{\sigma}'\} = [\mathbf{D}']\{\boldsymbol{\epsilon}\} \tag{6.77}$$

and

$$\{\mathbf{u}\} = [\mathbf{D}_u]\{\boldsymbol{\epsilon}\} \tag{6.78}$$

Combining these equations gives,

$$\{\sigma\} = [\mathbf{D}]\{\boldsymbol{\epsilon}\} \tag{6.79}$$

where

$$[\mathbf{D}] = [\mathbf{D}'] + [\mathbf{D}_u] \tag{6.80}$$

The matrix $[\mathbf{D}']$ is the familiar elastic stress–strain matrix in terms of effective Young's modulus E' and Poisson's ratio v' from (2.77). The matrix $[\mathbf{D}_u]$ contains the apparent bulk modulus of the fluid K_e in the following locations:

$$[\mathbf{D}_u] = \begin{bmatrix} K_e & K_e & 0 & K_e \\ K_e & K_e & 0 & K_e \\ 0 & 0 & 0 & 0 \\ K_e & K_e & 0 & K_e \end{bmatrix} \tag{6.81}$$

assuming that the third column corresponds to the shear terms in a two-dimensional plane–strain analysis.

To implement this method in the programs described in this chapter, it is necessary to form the global stiffness matrix using the total stress–strain matrix $[\mathbf{D}]$, while effective stresses for use in the failure function are computed from total strains using the effective stress–strain matrix $[\mathbf{D}']$. Pore pressures are simply computed from:

$$\{\mathbf{u}\} = K_e\,(\{\boldsymbol{\epsilon}_r\} + \{\boldsymbol{\epsilon}_z\} + \{\boldsymbol{\epsilon}_\theta\}) \begin{Bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{Bmatrix} \tag{6.82}$$

assuming an axisymmetric analysis.

For relatively large values of K_e, the analysis is insensitive to the exact magnitude of K_e. For axisymmetric analyses, Griffiths (1985) defined the dimensionless group,

$$\beta_t = \frac{(1-2v')K_e}{E'} \tag{6.83}$$

and showed that for an undrained triaxial test in a non-dilative material ($\psi = 0$), consolidated at a cell pressure of σ_3, the deviator stress at failure would be given by,

$$D_f = \frac{\sigma_3(K_p - 1)(3\beta_t + 1)}{(K_p + 2)\beta_t + 1} \tag{6.84}$$

where $K_p = \tan^2(45^\circ + \phi'/2)$.

In the limit as $\beta_t \to \infty$, this expression tends to,

$$D_f = \frac{3\sigma_3(K_p - 1)}{(K_p + 2)} \tag{6.85}$$

although for numerical purposes, undrained behaviour is essentially captured for $\beta_t > 20$

Program 6.9 Axisymmetric "undrained" strain of an elastic–plastic (Mohr–Coulomb) solid using 8-node rectangular quadrilaterals. Viscoplastic strain method.

```
PROGRAM p69
!-------------------------------------------------------------------------
! Program 6.9 Axisymmetric 'undrained' strain of an elastic-plastic
!             (Mohr-Coulomb) solid using 8-node rectangular
!             quadrilaterals. Viscoplastic strain method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,incs,iters,iy,k,limit,ndim=2,ndof=16,nels, &
   neq,nip=4,nn,nod=8,nodof=2,nr,nst=4,nxe,nye
 REAL(iwp)::bulk,c,cons,det,dq1,dq2,dq3,dsbar,dt,d4=4.0_iwp,              &
   d180=180.0_iwp,e,f,lode_theta,one=1.0_iwp,phi,pi,presc,psi,sigm,snph,  &
   penalty=1.e20_iwp,tol,two=2.0_iwp,v,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:),no(:),    &
   node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   der(:,:),deriv(:,:),devp(:),eld(:),eload(:),eps(:),erate(:),           &
   etensor(:,:,:),evp(:),evpt(:,:,:),flow(:,:),fun(:),gc(:),g_coord(:,:), &
   jac(:,:),km(:,:),kv(:),loads(:),m1(:,:),m2(:,:),m3(:,:),oldis(:),      &
   points(:,:),pore(:,:),sigma(:),storkv(:),stress(:),tensor(:,:,:),      &
   totd(:),weights(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,phi,c,psi,e,v,bulk,cons
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   x_coords(nxe+1),y_coords(nye+1),num(nod),evpt(nst,nip,nels),           &
   coord(nod,ndim),g_g(ndof,nels),tensor(nst,nip,nels),fun(nod),          &
   etensor(nst,nip,nels),dee(nst,nst),pore(nip,nels),stress(nst),         &
   jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),          &
   bee(nst,ndof),km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof), &
   eload(ndof),erate(nst),evp(nst),devp(nst),g(ndof),m1(nst,nst),         &
   m2(nst,nst),m3(nst,nst),flow(nst,nst),gc(ndim))
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq))
!-----------------------loop the elements to find global arrays sizes-----
 kdiag=0
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I7))')                                                &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------add fluid bulk modulus effective dee matrix-------
 CALL deemat(dee,e,v); pi=ACOS(-one)
 DO i=1,nst; DO k=1,nst; IF(i/=3.AND.k/=3)dee(i,k)=dee(i,k)+bulk
   END DO; END DO; snph=SIN(phi*pi/d180)
 dt=d4*(one+ v)*(one-two*v)/(e*(one-two*v+snph*snph))
 CALL sample(element,points,weights); kv=zero; tensor=zero; etensor=zero
!-----------------------element stiffness integration and assembly--------
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i)
     CALL bee8(bee,coord,points(i,1),points(i,2),det)
     gc=MATMUL(fun,coord); bee(4,1:ndof-1:2)=fun(:)/gc(1)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)*gc(1)
     tensor(1:2,i,iel)=cons; tensor(4,i,iel)=cons
   END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
 END DO elements_2
!-----------------------read displacement data and factorise equations----
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),no(fixed_freedoms),&
     storkv(fixed_freedoms))
   READ(10,*)(node(i),sense(i),i=1,fixed_freedoms)
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   kv(kdiag(no))=kv(kdiag(no))+penalty; storkv=kv(kdiag(no))
 END IF; CALL sparin(kv,kdiag); CALL deemat(dee,e,v)
!-----------------------displacement increment loop-----------------------
 READ(10,*)tol,limit,incs,presc
 WRITE(11,'(/A)')"  step   disp      dev stress  pore press  iters"
 oldis=zero; totd=zero
 disp_incs: DO iy=1,incs
   iters=0; bdylds=zero; evpt=zero
!-----------------------plastic iteration loop----------------------------
   its: DO
     iters=iters+1
     WRITE(*,'(A,E11.3,A,I4)')"  displacement",iy*presc,"  iteration",iters
     loads=zero; loads(no)=storkv*presc; loads=loads+bdylds
     CALL spabac(kv,loads,kdiag)
!-----------------------check plastic convergence-------------------------
     CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
!-----------------------go round the Gauss points ------------------------
     elements_3: DO iel=1,nels
       bload=zero; num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
       g=g_g(:,iel); eld=loads(g)
       gauss_pts_2: DO i=1,nip
         CALL shape_fun(fun,points,i)
         CALL bee8(bee,coord,points(i,1),points(i,2),det)
         gc=MATMUL(fun,coord); bee(4,1:ndof-1:2)=fun(:)/gc(1)
         eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel); sigma=MATMUL(dee,eps)
         stress=sigma+tensor(:,i,iel)
         CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
         IF(f>=zero)THEN
```

```
          CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
          CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
          erate=MATMUL(flow,stress); evp=erate*dt
          evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
          eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)*gc(1)
        END IF
!------update the Gauss Point stresses and calculate pore pressures-------
        IF(converged.OR.iters==limit)THEN
          tensor(:,i,iel)=stress
          etensor(:,i,iel)=etensor(:,i,iel)+eps+evpt(:,i,iel)
          pore(i,iel)=(etensor(1,i,iel)+etensor(2,i,iel)+            &
            etensor(4,i,iel))*bulk
        END IF
      END DO gauss_pts_2
!-----------------------compute the total bodyloads vector ---------------
      bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
    END DO elements_3; IF(converged.OR.iters==limit)EXIT
  END DO its; totd=totd+loads
  WRITE(11,'(I5,3E12.4,I5)')iy,totd(no(1)),dsbar, pore(1,1),iters
  IF(iters==limit)EXIT
 END DO disp_incs
 CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p69
```

New scalar reals:

`bulk`	apparent fluid bulk modulus (K_e)
`cons`	consolidating stress (σ_3)

New dynamic real arrays:

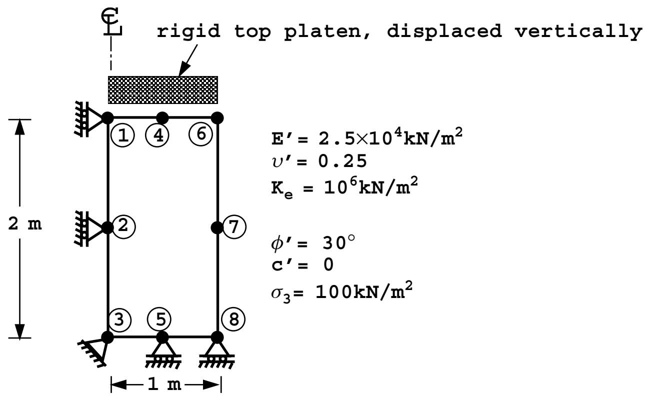
`etensor`	holds running total of all integrating point strain terms
`pore`	holds running total of all integrating point pore pressures

The example shown in Figure 6.42 represents a single axisymmetric 8-node element subjected to vertical compressive displacement increments along its top face. The analysis is of a “CU” triaxial test, in which the sample has been consolidated under a cell pressure of $\sigma_3 = 100$ kN/m^2, followed by undrained axial loading. In order to compute pore pressures during undrained loading, it is necessary to update strains (`etensor`) as well as stresses after each increment. The pore pressure is also computed from equation (6.82).

This program assumes a homogeneous material described by a Mohr–Coulomb failure criterion. In addition to the usual shear strength, dilation, and elastic parameters, the data file must provide the apparent fluid bulk modulus `bulk` and the initial consolidation stress `cons`

After the effective stress–strain matrix has been augmented by the fluid bulk modulus according to (6.80), the global stiffness matrix is formed in the usual way. Prescribed axial displacement increments are then applied to the top of the element using the “penalty” method as used previously in Program 6.4.

Just before the displacement increment loop begins, the subroutine `deemat` is called to form the effective stress–strain matrix. The data shown in Figure 6.42 is for an undrained



```
nxe  nye
1    1

prop(phi,c,psi,e,v,ke)
30.0  0.0  0.0  2.5e4  0.25  1.0e6

cons
 100.0

x_coords y_coords
0.0   1.0
0.0   2.0

nr,(k,nf(:,k),i=1,nr)
5
1 0 1 -2 0 1  3 0 0  5 1 0  8 1 0

fixed_freedoms,(node(i),sense(i),i=1,fixed_freedoms)
3
1  2   4  2   6  2

tol      limit  incs   presc
0.0001    50     8    -2.0e-3
```

Figure 6.42 Mesh and data for Program 6.9 example

"sand" with the following properties:

$$\begin{aligned}
\phi' &= 30^\circ \\
c' &= 0 \\
E' &= 2.5 \times 10^4 \text{ kN/m}^2 \\
\nu' &= 0.25 \\
K_e &= 10^6 \text{ kN/m}^2
\end{aligned}$$

The triaxial specimen has been consolidated under a compressive cell pressure of 100 kN/m^2 before undrained loading commences.

The output of two analyses is presented in Figure 6.43. In analysis (a), $\psi = 0$ and in analysis (b), $\psi = 30^\circ$. As expected, the inclusion of dilation has a considerable impact on the response in this "confined" problem. The deviator stress versus vertical deflection has been plotted for both undrained cases in Figure 6.44 together with the drained result obtained by setting $K_e = 0$. In case (a), where there is no plastic volume change ($\psi = 0$),

```
There are     10 equations and the skyline storage is     55

      step   disp      dev stress  pore press  iters
         1 -0.2000E-02  0.2990E+02 -0.9804E+01    2
         2 -0.4000E-02  0.5980E+02 -0.1961E+02    2
         3 -0.6000E-02  0.8971E+02 -0.2941E+02    2
         4 -0.8000E-02  0.1196E+03 -0.3922E+02    2
         5 -0.1000E-01  0.1209E+03 -0.3965E+02    4
         6 -0.1200E-01  0.1210E+03 -0.3966E+02    4
         7 -0.1400E-01  0.1210E+03 -0.3966E+02    4
         8 -0.1600E-01  0.1210E+03 -0.3966E+02    4
```

(a) $\psi = 0$

```
There are     10 equations and the skyline storage is     55

      step   disp      dev stress  pore press  iters
         1 -0.2000E-02  0.2990E+02 -0.9804E+01    2
         2 -0.4000E-02  0.5980E+02 -0.1961E+02    2
         3 -0.6000E-02  0.8971E+02 -0.2941E+02    2
         4 -0.8000E-02  0.1196E+03 -0.3922E+02    2
         5 -0.1000E-01  0.1299E+03 -0.2960E+02    3
         6 -0.1200E-01  0.1439E+03 -0.2363E+02    3
         7 -0.1400E-01  0.1570E+03 -0.1683E+02    3
         8 -0.1600E-01  0.1703E+03 -0.1022E+02    3
```

(b) $\psi = 30°$

Figure 6.43 Results from Program 6.9 example with (a) $\psi = 0$ and (b) $\psi = 30^\circ$

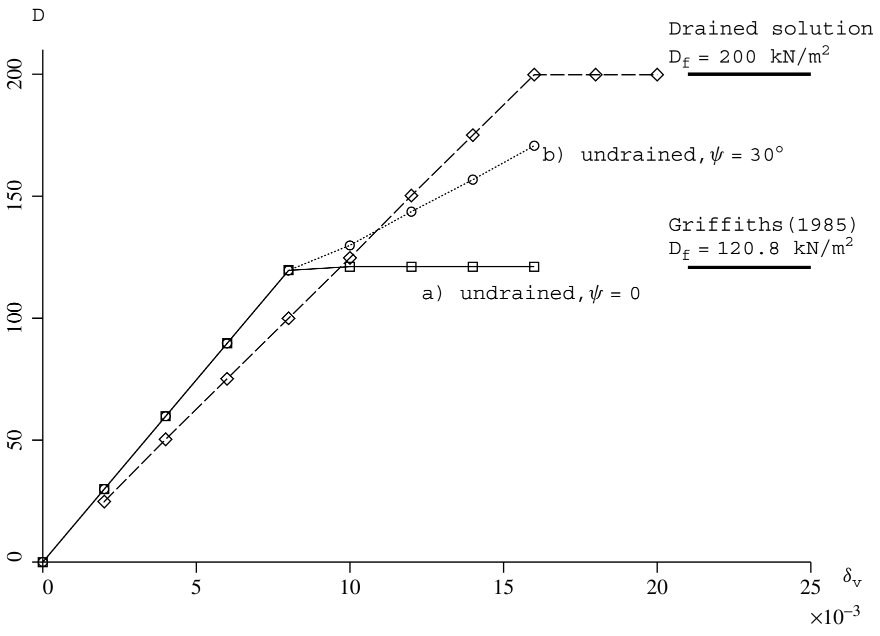


Figure 6.44 Vertical displacement versus deviator stress for drained and undrained triaxial loading. (a) $\psi = 0$ and (b) $\psi = 30^\circ$

the deviator stress reaches a peak of 121 kN/m^2, which is in close agreement with the closed form solution of 120.8 kN/m^2 (Griffiths, 1985) given by (6.84) for this problem with $\beta_t = 20$.

The deviator stress at failure in case (a) is significantly smaller than the drained value of 200 kN/m^2 due to the compressive pore pressures generated during elastic compression. Case (b), which includes an associated flow rule ($\psi = \phi' = 30°$), shows no sign of failure due to the tendency for dilation. In this case, the pore pressures continue to fall and the deviator stress continues to rise. This trend would continue indefinitely unless some additional criterion (e.g. cavitation or particle crushing) was introduced. It may also be noted from Figure 6.44, that the undrained response is slightly stiffer than the drained response at small strains.

While the presence of dilation has a very significant influence on undrained behaviour, it has little influence on the deviator stress at failure in the drained case.

Program 6.10 Three-dimensional strain analysis of an elastic–plastic (Mohr–Coulomb) slope using 20-node hexahedra. Viscoplastic strain method.

```
 PROGRAM p610
!-------------------------------------------------------------------------
! Program 6.10 Three-dimensional strain analysis of an elastic-plastic
!              (Mohr-Coulomb) slope using 20-node hexahedra. Viscoplastic
!              strain method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,ifix,iters,iy,limit,ndim=3,ndof=60,nels,neq,nip=8,nn,     &
   nod=20,nodof=3,nprops=6,np_types,nsrf,nst=6,nx1,nx2,ny1,ny2,nze
 REAL(iwp)::cf,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,d1,d4=4.0_iwp,     &
   d180=180.0_iwp,e,f,fmax,h1,h2,lode_theta,one=1.0_iwp,phi,phif,pi,psi,  &
   psif,sigm,snph,start_dt=1.e15_iwp,s1,tnph,tnps,tol,two=2.0_iwp,v,w1,w2,&
   zero=0.0_iwp
 CHARACTER(LEN=80)::element='hexahedron'; LOGICAL::converged
!----------------------dynamic arrays-------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   num(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   der(:,:),deriv(:,:),devp(:),eld(:),eload(:),eps(:),erate(:),evp(:),    &
   evpt(:,:,:),flow(:,:),fun(:),gravlo(:),g_coord(:,:),jac(:,:),km(:,:),  &
   kv(:),loads(:),m1(:,:),m2(:,:),m3(:,:),oldis(:),points(:,:),prop(:,:), &
   sigma(:),srf(:),weights(:)
!----------------------input and initialisation---------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res');
!---(ifix=1) smooth-smooth; (ifix=2) rough-smooth; (ifix=3) rough-rough---
 READ(10,*)w1,s1,w2,h1,h2,d1,nx1,nx2,ny1,ny2,nze,ifix,np_types
 nels=(nx1*ny1+ny2*(nx1+nx2))*nze
 nn=((3*(ny1+ny2)+2)*nx1+2*(ny1+ny2)+1+(3*ny2+2)*nx2)*(1+nze)+            &
   ((ny1+ny2+1)*(nx1+1)+(ny2+1)*nx2)*nze
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod),     &
   g_g(ndof,nels),jac(ndim,ndim),der(ndim,nod),etype(nels),               &
   deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof), &
   eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst),       &
   devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),   &
   prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 CALL emb_3d_bc(ifix,nx1,nx2,ny1,ny2,nze,nf); neq=MAXVAL(nf)
 ALLOCATE(loads(0:neq),bdylds(0:neq),oldis(0:neq),gravlo(0:neq),kdiag(neq))
!-----------------------loop the elements to find global arrays sizes-----
kdiag=0
 elements_1: DO iel=1,nels
   CALL emb_3d_geom(iel,nx1,nx2,ny1,ny2,nze,w1,s1,w2,h1,h2,d1,coord,num)
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(A,I7,A,I8)')                                                 &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); kv=zero; gravlo=zero
!-----------------------element stiffness integration and assembly--------
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; eld=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
     eld(nodof:ndof:nodof)=eld(nodof:ndof:nodof)+fun(:)*det*weights(i)
   END DO gauss_pts_1
   CALL fsparv(kv,km,g,kdiag); gravlo(g)=gravlo(g)-eld*prop(4,etype(iel))
 END DO elements_2
!-----------------------factorise equations------------------------------
 CALL sparin(kv,kdiag); pi=ACOS(-one)
!-----------------------trial strength reduction factor loop--------------
 READ(10,*)tol,limit,nsrf; ALLOCATE(srf(nsrf)); READ(10,*)srf
 WRITE(11,'(/A)')"    srf   max disp  iters"
 srf_trials: DO iy=1,nsrf
   dt=start_dt
   DO i=1,np_types
     phi=prop(1,i); tnph=TAN(phi*pi/d180); phif=ATAN(tnph/srf(iy))
     snph=SIN(phif); e=prop(5,i); v=prop(6,i)
     ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
   END DO; iters=0; bdylds=zero; evpt=zero; oldis=zero
!-----------------------plastic iteration loop---------------------------
   its: DO
     fmax=zero; iters=iters+1; loads=gravlo+bdylds
     CALL spabac(kv,loads,kdiag); loads(0)=zero
!-----------------------check plastic convergence------------------------
     CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
     IF(converged.OR.iters==limit)bdylds=zero
!-----------------------go round the Gauss Points ------------------------
     elements_3: DO iel=1,nels
       bload=zero; phi=prop(1,etype(iel)); tnph=TAN(phi*pi/d180)
       phif=ATAN(tnph/srf(iy))*d180/pi; psi=prop(3,etype(iel))
       tnps=TAN(psi*pi/d180); psif=ATAN(tnps/srf(iy))*d180/pi
       cf=prop(2,etype(iel))/srf(iy); e=prop(5,etype(iel))
       v=prop(6,etype(iel)); CALL deemat(dee,e,v); num=g_num(:,iel)
       coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
       gauss_points_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
         eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
         sigma=MATMUL(dee,eps); CALL invar(sigma,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         CALL mocouf(phif,cf,sigm,dsbar,lode_theta,f); IF(f>fmax)fmax=f
         IF(converged.OR.iters==limit)THEN; devp=sigma; ELSE
           IF(f>=zero.OR.(converged.OR.iters==limit))THEN
             CALL mocouq(psif,dsbar,lode_theta,dq1,dq2,dq3)
             CALL formm(sigma,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
             erate=MATMUL(flow,sigma); evp=erate*dt
             evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
           END IF
         END IF
         IF(f>=zero.OR.(converged.OR.iters==limit))THEN
           eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
         END IF
       END DO gauss_points_2
!-----------------------compute the total bodyloads vector----------------
       bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
     END DO elements_3
     WRITE(*,'(A,F7.2,A,I4,A,F8.3)')                                     &
       "  srf",srf(iy),"  iteration",iters,"  F_max",fmax
     IF(converged.OR.iters==limit)EXIT
   END DO its; WRITE(11,'(F7.2,E12.4,I5)')srf(iy),MAXVAL(ABS(loads)),iters
   IF(iters==limit)EXIT
 END DO srf_trials
STOP
END PROGRAM p610
```

New scalar integers:

`ifix`	sets boundary conditions, (1 smooth-smooth, 2 rough-smooth, 3 rough-rough)
`nze`	number of elements in z-direction

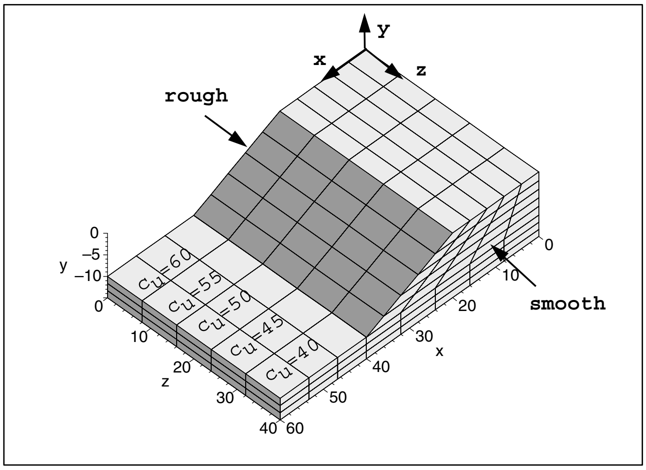
New scalar reals:

`d1`	depth of mesh in z-direction

This program demonstrates 3D plasticity analysis using 20-noded hexahedral elements with "reduced" (`nip=8`) integration. The example is of a simple 3D slope stability analysis, and the program is very similar to its 2D counterpart Program 6.3. Only three additional inputs are required as compared with the data for Program 6.3. The first is `ifix` which fixes the front and back faces of the mesh (in the z-direction) to either "rough" or "smooth". When `ifix=1`, both boundaries are smooth, and if the slope is homogeneous the analysis essentially reduces to plane strain, when `ifix=2` the front face is rough and the back face smooth, implying a line of symmetry along the centre of the embankment, and when `ifix=3` both boundaries are rough, enabling a full 3D analysis of a slope that may have non-uniform and non-symmetric properties in the crest (z-) direction. The second new input parameter is `nze`, which defines the number of slices of elements required in the z-direction, and the third is `d1` which represents the depth of the slope in the z-direction.

Two new subroutines, `emb_3d_bc` and `emb_3d_geom` are introduced to generate, respectively, the nodal freedom array `nf`, and the nodal coordinates and element node numbering `g_coord` and `g_num`. The subroutines create a rather simple 3D geometry in which the 2D cross-section (see Figure 6.15) is extrapolated uniformly in the z-direction. Users are invited to introduce their own mesh-generation techniques to study more realistic 3D geometries.

The slope and data to be considered are shown in Figure 6.45. The figure shows a 2:1 slope consisting of an embankment of height 10 m resting on a foundation of depth 5 m. The



```
w1     s1     w2     h1      h2    d1
20.0  20.0  20.0  10.0    5.0  40.0

nx1  nx2  ny1  ny2  nze
5    3    5    3    5

ifix  np_types
2       5

prop(phi,c,psi,gamma,e,v)
0.0 60.0  0.0  20.0  1.0e5  0.3
0.0 55.0  0.0  20.0  1.0e5  0.3
0.0 50.0  0.0  20.0  1.0e5  0.3
0.0 45.0  0.0  20.0  1.0e5  0.3
0.0 40.0  0.0  20.0  1.0e5  0.3

etype(i),i=1,nels  (x then y then z)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

tol   limit
0.0001 1000

nsrf,(srf(i),i=1,nsrf)
6
1.0 1.4 1.5 1.55 1.58 1.60
```

Figure 6.45 Mesh and data for Program 6.10 example

depth of the mesh in the z-direction is 40 m `d1=40`, and the boundary condition parameter is set to `ifix=2` implying a line of symmetry at the $z = 40$ m plane (rough-smooth), so the "actual" embankment has a depth of 80 m.

Three-dimensional analysis involves very significant storage requirements compared with 2D, so the example involves a quite crude mesh with 5 slices of elements in the z-direction `nze=5`. In this case the slope is assumed to have a linearly varying undrained strength varying from 60 kN/m^2 at the abutment to 40 kN/m^2 at the centreline. There are thus 5 property types (`np_types=5`) in the data file, one for each slice.

The `etype` data maps the properties onto the elements, which are numbered, starting at the origin, first in the x-direction, then in the y-direction (top to bottom) and finally in the z-direction. The tolerance `tol` and iteration ceiling `limit` are set as usual, followed by the number of trial strength reduction factors `nsrf` and the strength reduction factor values read into `srf`. The iteration ceiling has been set higher than usual at 1,000 to emphasise the onset of failure.

The output from the analysis is given in Figure 6.46 and plotted in Figure 6.47. The results indicate that the factor of safety of the slope is about 1.6. Figure 6.48 shows the

```
There are   2972 equations and the skyline storage is 1538004

   srf    max disp   iters
  1.00   0.2266E-01    16
  1.40   0.3123E-01    77
  1.50   0.3930E-01   208
  1.55   0.4971E-01   380
  1.58   0.6363E-01   703
  1.60   0.9308E-01  1000
```

Figure 6.46 Results from Program 6.10 example

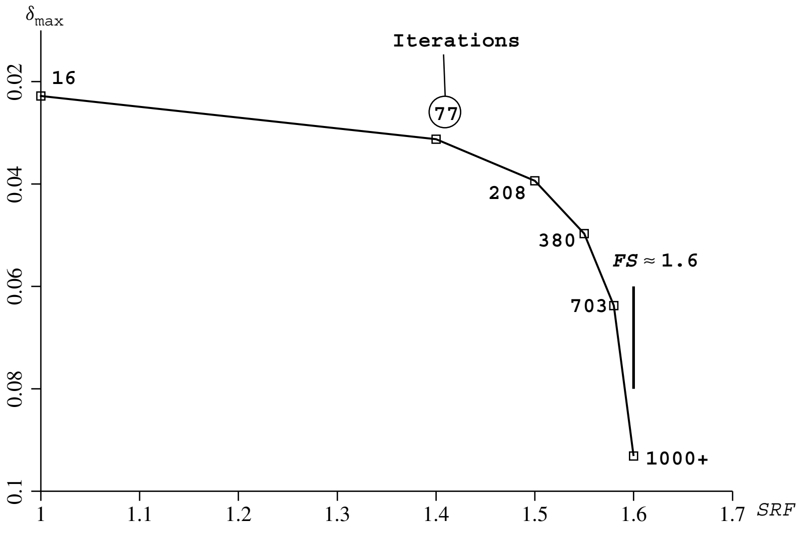


Figure 6.47 Plot of maximum displacement versus Strength Reduction Factor from Program 6.10 example

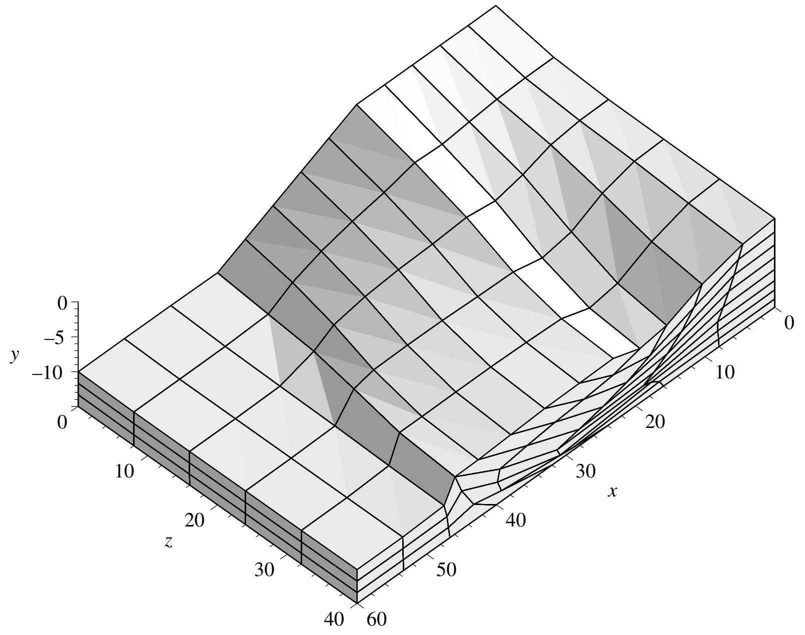


Figure 6.48 Deformed mesh at failure from Program 6.10 example

deformed mesh corresponding to this unconverged solution. The slumping of the slope towards its centreline is clearly seen.

Program 6.11 Three-dimensional strain analysis of an elastic–plastic (Mohr–Coulomb) slope using 20-node hexahedra. Viscoplastic strain method. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver.

```
PROGRAM p611
!-------------------------------------------------------------------------
! Program 6.11 Three-dimensional strain analysis of an elastic-plastic
!              (Mohr-Coulomb) slope using 20-node hexahedra. Viscoplastic
!              strain method. No global stiffness matrix assembly.
!              Diagonally preconditioned conjugate gradient solver.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,cg_tot,i,iel,ifix,iters,iy,k,limit,ndim=3,    &
   ndof=60,nels,neq,nip=8,nn,nod=20,nodof=3,nprops=6,np_types,nsrf,nst=6, &
   nx1,nx2,ny1,ny2,nze
 REAL(iwp)::alpha,beta,cf,cg_tol,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp, &
   d1,d4=4.0_iwp,d180=180.0_iwp,e,f,fmax,h1,h2,lode_theta,one=1.0_iwp,phi,&
   phif,pi,psi,psif,sigm,snph,start_dt=1.e15_iwp,s1,tnph,tnps,tol,        &
   two=2.0_iwp,up,v,w1,w2,zero=0.0_iwp
 CHARACTER(LEN=80)::element='hexahedron'; LOGICAL::converged,cg_converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),nf(:,:),num(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),d(:),      &
   dee(:,:),der(:,:),deriv(:,:),devp(:),diag_precon(:),eld(:),eload(:),   &
   eps(:),erate(:),evp(:),evpt(:,:,:),flow(:,:),fun(:),gravlo(:),         &
   g_coord(:,:),jac(:,:),km(:,:),loads(:),m1(:,:),m2(:,:),m3(:,:),        &
   oldis(:),p(:),points(:,:),prop(:,:),sigma(:),srf(:),storkm(:,:,:),u(:),&
   weights(:),x(:),xnew(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
!---(ifix=1) smooth-smooth; (ifix=2) rough-smooth; (ifix=3) rough-rough---
 READ(10,*)w1,s1,w2,h1,h2,d1,nx1,nx2,ny1,ny2,nze,ifix,                    &
   cg_tol,cg_limit,np_types
 nels=(nx1*ny1+ny2*(nx1+nx2))*nze
 nn=((3*(ny1+ny2)+2)*nx1+2*(ny1+ny2)+1+(3*ny2+2)*nx2)*(1+nze)+            &
   ((ny1+ny2+1)*(nx1+1)+(ny2+1)*nx2)*nze
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod),     &
   g_g(ndof,nels),jac(ndim,ndim),der(ndim,nod),etype(nels),               &
   deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof), &
   eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst),       &
   devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),   &
   prop(nprops,np_types),storkm(ndof,ndof,nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 CALL emb_3d_bc(ifix,nx1,nx2,ny1,ny2,nze,nf); neq=MAXVAL(nf)
 WRITE(11,'(A,I7,A)')" There are",neq," equations"
 ALLOCATE(loads(0:neq),bdylds(0:neq),oldis(0:neq),gravlo(0:neq),p(0:neq), &
   x(0:neq),xnew(0:neq),u(0:neq),diag_precon(0:neq),d(0:neq))
!-----------------------loop the elements to find global array sizes-----
 elements_1: DO iel=1,nels
   CALL emb_3d_geom(iel,nx1,nx2,ny1,ny2,nze,w1,s1,w2,h1,h2,d1,coord,num)
   g_num(:,iel)=num; CALL num_to_g(num,nf,g)
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1
 CALL sample(element,points,weights); diag_precon=zero; gravlo=zero
!----------element stiffness integration, storage and preconditioner------
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; eld=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
     eld(nodof:ndof:nodof)=eld(nodof:ndof:nodof)+fun(:)*det*weights(i)
   END DO gauss_pts_1; storkm(:,:,iel)=km
   DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
   gravlo(g)=gravlo(g)-eld*prop(4,etype(iel))
 END DO elements_2; diag_precon(1:)=one/diag_precon(1:); pi=ACOS(-one)
!-----------------------trial strength reduction factor loop--------------
 READ(10,*)tol,limit,nsrf; ALLOCATE(srf(nsrf)); READ(10,*)srf
 WRITE(11,'(/A)')"    srf   max disp  iters     cg iters/plastic iter"
 srf_trials: DO iy=1,nsrf
   dt=start_dt
   DO i=1,np_types
     phi=prop(1,i); tnph=TAN(phi*pi/d180); phif=ATAN(tnph/srf(iy))
     snph=SIN(phif); e=prop(5,i); v=prop(6,i)
     ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
   END DO; iters=0; bdylds=zero; evpt=zero; oldis=zero;
   cg_tot=0; diag_precon(0)=zero
!-----------------------plastic iteration loop----------------------------
   its: DO
```

```
     iters=iters+1; loads=gravlo+bdylds; d=diag_precon*loads; p=d; x=zero;
     cg_iters=0; fmax=zero
!-----------------------pcg equation solution----------------------------
     pcg: DO
       cg_iters=cg_iters+1; u=zero
       elements_3 : DO iel=1,nels
         CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
         g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
       END DO elements_3
       up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
       loads=loads-u*alpha; d=diag_precon*loads
       beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
       CALL checon(xnew,x,cg_tol,cg_converged)
       IF(cg_converged.OR.cg_iters==cg_limit)EXIT
     END DO pcg; cg_tot=cg_tot+cg_iters; loads=xnew; loads(0)=zero
!-----------------------check plastic convergence------------------------
     CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
     IF(converged.OR.iters==limit)bdylds=zero
!-----------------------go round the Gauss Points -----------------------
     elements_4: DO iel=1,nels
       bload=zero; phi=prop(1,etype(iel)); tnph=TAN(phi*pi/d180)
       phif=ATAN(tnph/srf(iy))*d180/pi; psi=prop(3,etype(iel))
       tnps=TAN(psi*pi/d180); psif=ATAN(tnps/srf(iy))*d180/pi
       cf=prop(2,etype(iel))/srf(iy); e=prop(5,etype(iel))
       v=prop(6,etype(iel)); CALL deemat(dee,e,v); num=g_num(:,iel)
       coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
       gauss_points_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
         eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
         sigma=MATMUL(dee,eps); CALL invar(sigma,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         CALL mocouf(phif,cf,sigm,dsbar,lode_theta,f); IF(f>fmax)fmax=f
         IF(converged.OR.iters==limit)THEN; devp=sigma; ELSE
           IF(f>=zero.OR.(converged.OR.iters==limit))THEN
             CALL mocouq(psif,dsbar,lode_theta,dq1,dq2,dq3)
             CALL formm(sigma,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
             erate=MATMUL(flow,sigma); evp=erate*dt
             evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
           END IF
         END IF
         IF(f>=zero.OR.(converged.OR.iters==limit))THEN
           eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
         END IF
       END DO gauss_points_2
!-----------------------compute the total bodyloads vector----------------
       bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
     END DO elements_4
     WRITE(*,'(A,F7.2,A,I4,A,F8.3)')                                       &
       "  srf",srf(iy),"  iteration",iters,"  F_max",fmax
     IF(converged.OR.iters==limit)EXIT
   END DO its; WRITE(11,'(F7.2,E12.4,I5,F17.2)')                           &
     srf(iy),MAXVAL(ABS(loads)),iters,REAL(cg_tot)/REAL(iters)
   IF(iters==limit)EXIT
 END DO srf_trials
STOP
END PROGRAM p611
```

The final program in this chapter repeats the analysis of Program 6.10, using a preconditioned conjugate gradient solver involving no mesh assembly. The rather crude mesh demonstrated in the previous example still required 1,538,004 locations in the vector `kv` in order to store the stiffness matrix using a skyline strategy. Even quite modest 3D meshes require arrays with millions of locations in order to store the global matrices.

The "mesh-free" approach made possible with preconditioned conjugate gradient (pcg) solvers, enables large 3D analyses to be performed on computers with modest core memory availability. Furthermore, the pcg approach is highly amenable to exploitation on computers with parallel architecture as will be demonstrated in detail in Chapter 12.

Program 6.11 (data shown in Figure 6.49) involves no new variables and can be considered an extension of Program 6.2, in which the pcg technique was first introduced in a plasticity analysis.

```
w1    s1    w2    h1    h2   d1
20.0  20.0  20.0  10.0  5.0  40.0

nx1  nx2  ny1  ny2  nze
5    3    5    3    5

ifix  cg_tol  cg_limit  np_types
2     0.0001  500       5

prop(phi,c,psi,gamma,e,v)
0.0 60.0  0.0  20.0  1.0e5  0.3
0.0 55.0  0.0  20.0  1.0e5  0.3
0.0 50.0  0.0  20.0  1.0e5  0.3
0.0 45.0  0.0  20.0  1.0e5  0.3
0.0 40.0  0.0  20.0  1.0e5  0.3

etype(i),i=1,nels  (x then y then z)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

tol   limit
0.0001 1000

nsrf,(srf(i),i=1,nsrf)
6
1.0 1.4 1.5 1.55 1.58 1.60
```

Figure 6.49 Data for Program 6.11 example

```
There are   2972 equations

   srf   max disp   iters     cg iters/plastic iter
  1.00  0.2264E-01    17           107.88
  1.40  0.3119E-01    78           110.40
  1.50  0.3918E-01   206           110.41
  1.55  0.4959E-01   379           110.11
  1.58  0.6377E-01   723           110.04
  1.60  0.9431E-01  1000           110.03
```

Figure 6.50 Results from Program 6.11 example

Two additional data values required are `pcg_tol` set to 0.0001, and `pcg_limit` set to 500. The output shown in Figure 6.50 is essentially identical to that shown in Figure 6.46 obtained using the direct solver.

Glossary of variable names used in Chapter 6

Scalar integers:

`cg_iters`	conjugate gradient iteration counter
`cg_limit`	conjugate gradient iteration ceiling
`cg_tot`	keeps running total of `cg_iters`
`fixed_freedoms`	number of fixed displacements
`i`	simple counter
`iel`	simple counter
`ifix`	sets 3D slope boundary conditions
`incs`	number of load increments
`iters`	counts plastic iterations
`iy`	counts load increments
`iwp`	`SELECTED_REAL_KIND(15)`
`k`	node number
`limit`	plastic iteration ceiling
`loaded_nodes`	number of loaded nodes
`ndim`	number of dimensions
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integrating points per element
`nn`	number of nodes is the mesh
`nod`	number of nodes per element
`nodof`	number of degrees of freedom per node
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes
`nsrf`	number of trial strength reduction factors
`nst`	number of stress (strain) terms
`nxe`	number of columns of elements in x-direction
`nx1`	number of columns of elements in embankment
`nx2`	number of columns of elements to right of toe
`nye`	number of rows of elements in y-direction
`ny1`	number of rows of elements in embankment
`ny2`	number of rows of elements in foundation
`nze`	number of slices of elements in z-direction

Scalar reals:

`alpha`	α from (3.22)
`beta`	β from (3.22)
`bot`	holds several dot products
`c`	cohesion

`bulk`	apparent fluid bulk modulus
`cf`	factored cohesion
`cg_tol`	pcg convergence tolerance
`cons`	consolidating stress (σ_3)
`c_e`	embankment cohesion
`c_f`	foundation cohesion
`dlam`	plastic multiplier λ
`ddt`	used to find the critical time step
`det`	determinant of the Jacobian matrix
`dq1`	plastic potential derivative, $\partial Q/\partial\sigma_m$
`dq2`	plastic potential derivative, $\partial Q/\partial J_2$
`dq3`	plastic potential derivative, $\partial Q/\partial J_3$
`dsbar`	invariant, $\overline{\sigma}$
`dslam`	plastic multiplier increment $\Delta\lambda$
`dt`	critical viscoplastic time step (set initially to 10^{15})
`d1`	depth of mesh in z-direction
`d3`	set to 3.0
`d4`	set to 4.0
`d180`	set to 180.0
`e`	Young's modulus
`enxe`	number of x-elements in embankment
`enye`	number of y-elements in embankment
`e_e`	Young's modulus in embankment
`e_f`	Young's modulus in foundation
`f`	value of yield function
`fac`	measure of yield surface overshoot(f from 6.35)
`ff`	holds a value of the yield function
`fftol`	tolerance on yield function
`fmax`	maximum value of yield function `F`
`fnew`	value of yield function after stress increment
`fnxe`	number of x-elements in foundation
`fnye`	number of y-elements in foundation
`fstiff`	holds a value of the yield function
`gama_e`	unit weight of embankment
`gama_f`	unit weight of foundation
`gamma`	soil unit weight
`h1`	height of embankment
`h2`	height of foundation
`ii`	counts the lifts
`itype`	type of degeneration of quadrilateral to triangle
`k0`	"at rest" earth pressure coefficient, K_o
`layers`	number of excavation steps
`lifts`	number of lifts
`lnn`	keeps running total of number of nodes
`load_theta`	Lode angle, θ
`ltol`	tolerance on `tloads`

`newele`	number of new elements at each lift
`noexe`	number of elements to be removed at each step
`nouts`	number of nodes at which output is required
`ntote`	holds running total of number of excavated elements
`oldele`	keeps running total of number of elements
`oldnn`	number of nodes from previous lift
`one`	set to 1.0
`ot`	overturning moment
`pav`	earth force based on stress averaging
`penalty`	set to 1×10^{20}
`phi`	friction angle (degrees)
`phif`	factored friction angle
`phi_e`	friction angle of embankment
`phi_f`	friction angle of foundation
`pi`	set to π
`pr`	earth force based on nodal reactions
`presc`	wall displacement increment
`psi`	dilation angle (degrees)
`psi_e`	dilation angle of embankment
`psi_f`	dilation angle of foundation
`psif`	factored dilation angle
`ptot`	holds running total of applied pressure
`pt5`	set to 0.5
`sigm`	mean stress, σ_m
`snph`	sin of `phi`
`start_dt`	starting value of `dt`
`s1`	width of top of embankment
`tloads`	holds the sum of `bdylds`
`tnph`	tangent of `phi`
`tnps`	tangent of `psi`
`tol`	plastic convergence tolerance
`top`	holds a dot product
`two`	set to 2.0
`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\{\mathbf{R}\}_k$ from (3.22)
`v`	Poisson's ratio
`v_e`	Poisson's ratio of embankment
`v_f`	Poisson's ratio of foundation
`w1`	width of sloping section of embankment
`w2`	distance foundation extends beyond the toe
`zero`	set to 0.0

Scalar character:

`element`	element type

Scalar logicals:

`cg_converged`	set to `.TRUE.` if pcg process has converged
`converged`	set to `.TRUE.` if plastic iterations have converged

Dynamic integer arrays:

`etype`	element property type vector
`exele`	element numbers of removed elements
`g`	element steering vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`kdiag`	diagonal term location vector
`nf`	nodal freedom matrix
`no`	fixed freedom numbers vector
`node`	loaded nodes vector
`num`	element node numbers vector
`sense`	sense of freedom to be fixed vector

Dynamic real arrays:

`acat`	used in development of (6.74)
`acatc`	used in development of (6.74)
`bdylds`	self-equilibrating global body loads
`bee`	strain-displacement matrix
`bload`	self-equilibrating element body loads
`caflow`	used in development of (6.74)
`coord`	element nodal coordinates
`d`	vector used in equation (3.22)
`daatd`	used in development of (6.74)
`ddylds`	global body loads
`dee`	stress–strain matrix
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`devp`	product $[\mathbf{D}^e]\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}$
`diag_precon`	diagonal preconditioner vector
`dl`	holds plastic multiplier λ for all Gauss points
`dload`	element body loads
`dsigma`	stress increment
`elastic`	elastic nodal displacements
`eld`	element nodal displacements
`eload`	integrating point contribution to `bload`
`elso`	plastic stresses
`eps`	strain terms
`erate`	viscoplastic strain rate, $\left\{\dot{\boldsymbol{\epsilon}}^{vp}\right\}$
`etensor`	holds running total of all integrating point strain terms
`evp`	viscoplastic strain rate increment, $\left\{\boldsymbol{\delta\epsilon}^{vp}\right\}$
`evpt`	holds running total of viscoplastic strains, $\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}$
`exc_loads`	excavation loads
`flow`	holds $\{\partial Q/\partial\boldsymbol{\sigma}\}$
`fun`	shape functions
`gc`	integrating point coordinates

`gravlo`	loads generated by gravity
`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`km`	element stiffness matrix
`kv`	global stiffness matrix
`loads`	nodal loads and displacements
`m1`	used to compute $\{\partial\sigma_m/\partial\boldsymbol{\sigma}\}$
`m2`	used to compute $\{\partial J_2/\partial\boldsymbol{\sigma}\}$
`m3`	used to compute $\{\partial J_3/\partial\boldsymbol{\sigma}\}$
`oldis`	nodal displacements from previous iteration
`p`	"descent" vector used in (3.22)
`pl`	plastic $[\mathbf{D}^p]$ matrix
`points`	integrating point local coordinates
`pore`	holds running total of all integrating point pore pressures
`prop`	element properties
`qinva`	used in development of (6.74)
`qinvr`	used in development of (6.74)
`qmat`	used in development of (6.74)
`react`	global nodal reaction forces
`ress`	used in development of (6.74)
`rmat`	used in development of (6.74)
`rload`	element nodal reaction forces
`qinc`	holds applied pressure increments
`sigma`	stress terms
`solid`	identifies "air" elements (= 0 for "air", = 1 for solid)
`srf`	trial strength reduction factors
`storkm`	holds element stiffness matrices
`storkv`	holds augmented stiffness diagonal terms
`stress`	stress term increments
`tensor`	holds running total of all integrating point stress terms
`totd`	holds running total of nodal displacements (vector)
`tot_d`	holds running total of nodal displacements (array)
`totex`	holds element numbers for all removed elements
`val`	applied nodal load weightings
`vmfl`	von Mises "flow" vector
`vmfla`	used in development of (6.74)
`vmflq`	used in development of (6.74)
`vmtemp`	used in development of (6.74)
`u`	vector used in equations (3.22)
`value`	fixed values of displacements
`weights`	weighting coefficients
`x`	"old" solution vector
`xnew`	"new" solution vector
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout

6.13 Exercises

1. Use Program 6.1 to estimate the bearing capacity of the surface strip footing shown in Figure 6.51 which is supported by undrained clay with a shear strength of $c_u = 50$ kN/m^2. (Ans: $q_{ult} \approx 257$ kN/m^2)

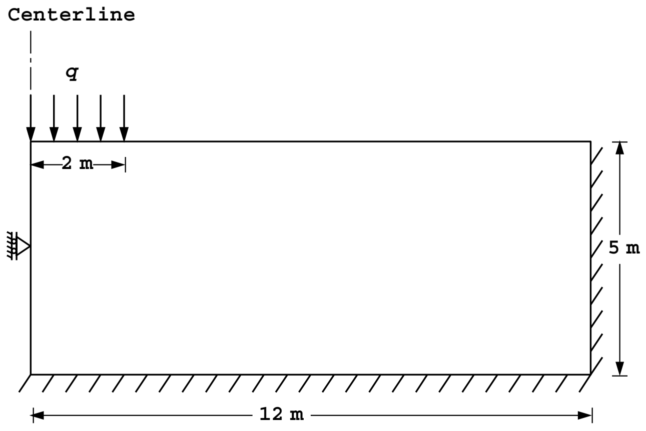


Figure 6.51

2. Repeat question 1 if the undrained shear strength increases linearly from 20 kN/m^2 at ground level to 50 kN/m^2 at 5 m depth. (Ans: $q_{ult} \approx 130$ kN/m^2)
3. Use Program 6.1 to estimate the bearing capacity of the footing shown in Figure 6.52 which is at the edge of a vertical cut of undrained clay with a shear strength of $c_u = 75$ kN/m^2. (Ans: $q_{ult} = 150$ kN/m^2)

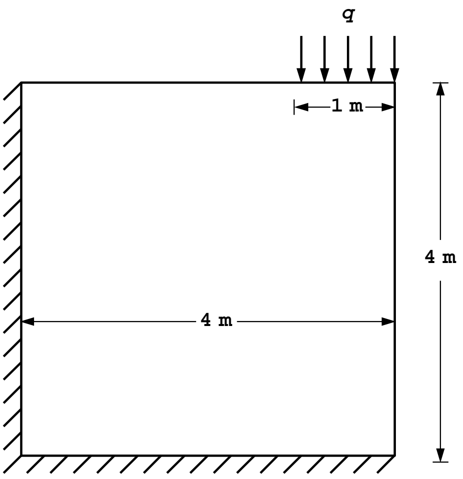


Figure 6.52

4. Use Program 6.3 to estimate the factor of safety of the slope shown in Figure 6.53. (Ans: $F \approx 1.45$)

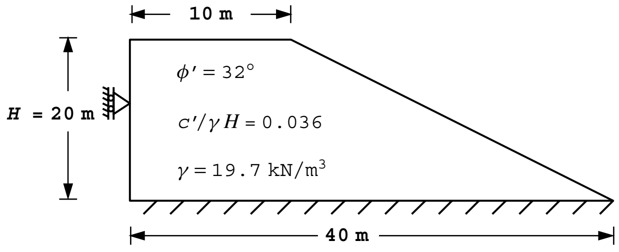


Figure 6.53

5. Use Program 6.3 to repeat the previous analysis if a second layer of soil is discovered in the lower part of the embankment as shown in Figure 6.54. (Ans: $F \approx 1.06$)

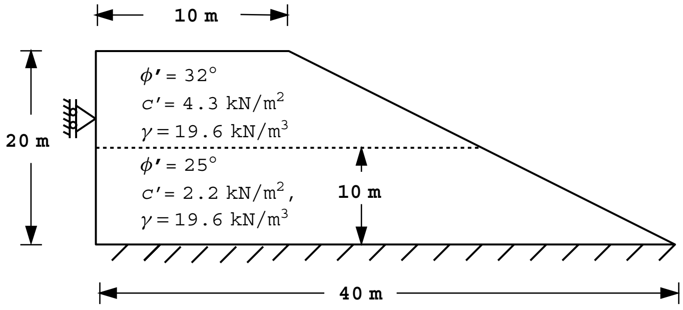


Figure 6.54

6. Use Program 6.3 to estimate the factor of safety of the slope shown in Figure 6.55 with $c_{u2} = 4, 7.5$ and $10\ \mathrm{kN/m^2}$. (Ans: $F \approx 1.22, 2.05, 2.09$)

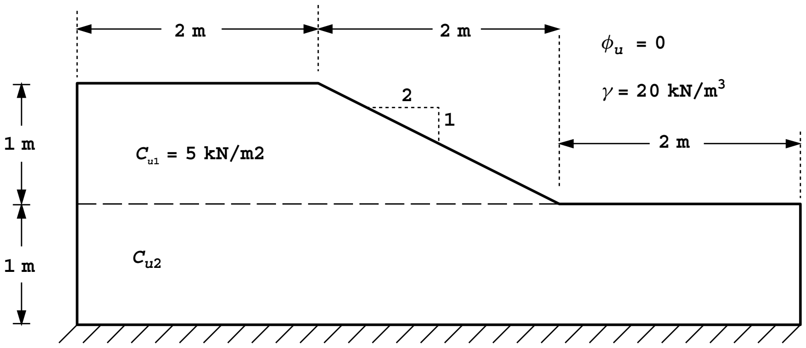


Figure 6.55

7. Use Program 6.4 to estimate the *active* force exerted by the soil on the wall shown in Figure 6.56. (Ans: $F \approx 3.3$ kN/m)

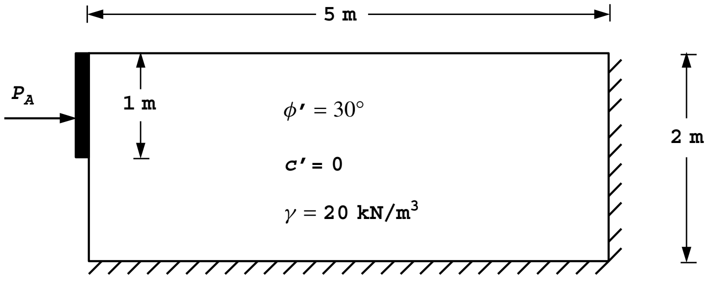


Figure 6.56

8. Use Program 6.8 to repeat the excavation example described in Figure 6.38 using the construction sequence of Cases D and E in Figure 6.37. (Ans: Case D fails after first excavation of elements 9 and 10. Case E fails after fourth excavation of element 14)

9. Use Program 6.10 or 6.11 to investigate the influence of the third dimension `d1` on the factor of safety of the cohesive slope shown in Figure 6.57. Gradually increase the depth `d1` of the mesh and use the symmetry boundary condition `ifix=2`.

(Ans:

d1	1.4	2.1	2.8	3.5	4.2	plane strain
F	1.97	1.75	1.66	1.59	1.56	1.45

)

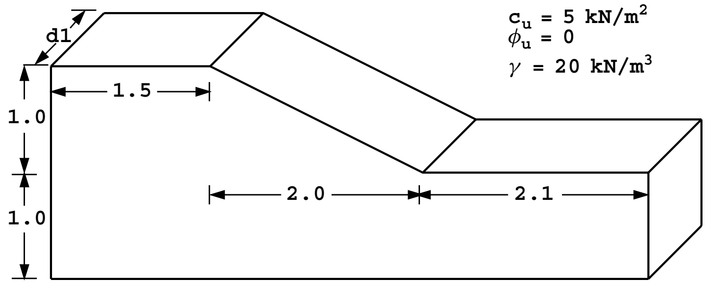


Figure 6.57

References

Bishop AW and Morgenstern NR 1960 Stability coefficients for earth slopes. *Géotechnique* **10**, 129–150.

Cormeau IC 1975 Numerical stability in quasi–static elasto–viscoplasticity. *Int J Numer Methods Eng* **9**(1), 109–127.

Duncan JM and Chang CY 1970 Non-linear analysis of stress and strain in soils. *J Soil Mech Found Div, ASCE* **96**(SM5), 1629–1653.

Griffiths DV 1980 *Finite Element Analyses of Walls, Footings and Slopes*. PhD thesis, Department of Engineering, University of Manchester.

Griffiths DV 1982 Computation of bearing capacity factors using finite elements. *Géotechnique* **32**(3), 195–202.

Griffiths DV 1985 The effect of pore fluid compressibility on failure loads in elasto-plastic soils. *Int J Numer Anal Methods Geomech* **9**, 253–259.

Griffiths DV and Lane PA 1999 Slope stability analysis by finite elements. *Géotechnique* **49**(3), 387–403.

Griffiths DV and Mustoe GGW 1995 Selective reduced integration of the four node plane element in closed-form. *J Eng Mech, ASCE* **121**(6), 725–729.

Griffiths DV and Willson SM 1986 An explicit form of the plastic matrix for a Mohr–Coulomb material. *Commun Appl Numer Methods* **2**, 523–529.

Hill R 1950 *The Mathematical Theory of Plasticity*. Oxford University Press.

Hughes TJR 1987 *The Finite Element Method*. Prentice-Hall, Englewood Cliffs, N.J.

Molenkamp F 1987 Kinematic model for alternating loading ALTERNAT. Technical report CO-218598, Delft Geotechnics, Delft.

Nayak GC and Zienkiewicz OC 1972 Elasto/Plastic stress analysis. A generalisation for various constitutive relationships including strain softening. *Int J Numer Methods Eng* **5**, 113–135.

Naylor DJ 1974 Stresses in nearly incompressible materials by finite elements with application to the calculation of excess pore pressure. *Int J Numer Methods Eng* **8**, 443–460.

Ortiz M and Popov EP 1985 Accuracy and stability integration algorithms for elasto-plastic constitutive relations. *Int J Numer Methods Eng* **21**, 1561–1576.

Rice JR and Tracey DM 1973 Computational fracture mechanics. In *Proceedings of Symposium on Numerical Methods and Structural Mechanics* (ed. Fenves S). Academic Press.

Smith IM 1997 Computation of large scale viscoplastic flows of frictional geotechnical materials. In *Dynamics of Complex Fluids* (ed. Adams MJ *et al*). Imperial College Press, pp. 425–445.

Smith IM and Ho DKH 1992 Influence of construction technique on performance of braced excavation in marine clay. *Int J Numer Anal Methods Geomech*, **16**, 845–867.

Smith IM and Hobbs R 1974 Finite element analysis of centrifuged and built-up slopes. *Géotechnique* **24**(4), 531–559.

Taylor DW 1937 Stability of earth slopes. *J Boston Soc Civil Eng* **24**, 197–246.

Yamada Y, Yoshimura N and Sakurai T 1968 Plastic stress-strain matrix and its application for the solution of elastic plastic problems by the finite element method. *J Mech Sci* **10**, 343–354.

Zienkiewicz OC and Cormeau IC 1974 Viscoplasticity, plasticity and creep in elastic solids. A unified approach. *Int J Numer Methods Eng* **8**, 821–845.

Zienkiewicz OC and Taylor RL 1989 *The Finite Element Method*, vol. 1, 4th edn. McGraw-Hill, London, New York.

Zienkiewicz OC, Humpheson C and Lewis RW 1975 Associated and non-associated viscoplasticity and plasticity in soil mechanics. *Géotechnique* **25**, 671–689.

Zienkiewicz OC, Valliappan S and King IP 1969 Elasto-plastic solutions of engineering problems, 'initial stress' finite element approach. *Int J Numer Methods Eng* **1**, 75–100.

7

Steady State Flow

7.1 Introduction

The five programs presented in this chapter solve steady state problems governed by Laplace's equation (2.122). Typical examples of this type of problem include steady seepage through soils and steady heat flow through a conductor. Examples are presented of planar (confined and unconfined), axisymmetric, and three-dimensional flow. Unlike the problems solved in Chapters 5 and 6, which gave vector fields of displacements, the dependent variable in these problems is a scalar, generically called the *potential* which may represent, for example, the total head in a seepage problem or the temperature in a heat flow analysis. Each node therefore has only one degree of freedom associated with it.

Systems that are governed by Laplace's equation require boundary conditions to be prescribed at all points around a closed domain. These boundary conditions commonly take the form of fixed values of the potential or its first derivative normal to the boundary. The problem amounts to finding the values of the potential at points within the closed domain.

Being "elliptic" in character, the solution of Laplace's equation quite closely resembles the solution of equilibrium equations (2.57) in solid elasticity. Both methods ultimately require the solution of a set of linear simultaneous equations. The element conductivity matrix (analogous to the "stiffness" matrix in elasticity) can be formed numerically, as described by equations (3.61) to (3.63) or "analytically" as discussed in Section 3.2.2. Either way, the element matrices can be assembled into a global conductivity matrix which, like its global elastic counterpart, is symmetrical, banded, and usually stored as a skyline. Alternatively, element-by-element iterative strategies can be used. Taking the analogy with Chapter 5 one stage further, "displacements" now become total heads and "loads" become net nodal inflow.

Program 7.1 describes the solution of Laplace's equation over a set of 1D elements, that can each have different lengths, areas, and permeabilities. The elements can be attached end to end, or in any desired "network" of connections. Program 7.2 describes the solution of Laplace's equation over a plane or axisymmetric 2D domain. Program 7.3 describes the non-linear problem of free-surface flow, in which the mesh is allowed to deform iteratively until it assumes the shape of the free surface at convergence. Program 7.4 is a general program for the solution of Laplace's equation over two- or three-dimensional domains. The final Program 7.5 describes an element-by-element version of Program 7.4 avoiding the need for global matrix assembly. A parallel version of this program is also described in Chapter 12.

As the problems in this chapter involve scalar fields with just one unknown at each node, the programming is simplified in that nod is always equal to ndof, so the latter is not required. A further change from the solid mechanics applications of the preceding chapters, is that the "zero freedoms" data previously introduced through the nf array has been removed. In this chapter, all fixed boundary conditions, whether equal to zero or not, are applied through the fixed_freedom data. Since all nodal freedoms are therefore retained in the assembly and analysis, nn is always equal to neq. In the interests of consistency with other programs in the book however, neq has been retained.

Program 7.1 One-dimensional analysis of steady seepage using 2-node line elements.

```
PROGRAM p71
!-------------------------------------------------------------------------
! Program 7.1 One dimensional analysis of steady seepage using
!             2-node line elements.
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,nels,neq,nod=2,nn,nprops=1, &
   np_types
 REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp
!-----------------------dynamic arrays-------------------------------------
 INTEGER,ALLOCATABLE::etype(:),kdiag(:),g_num(:,:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::disps(:),ell(:),kp(:,:),kv(:),kvh(:),loads(:),    &
   prop(:,:),value(:)
!-----------------------input and initialisation---------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,nn,np_types; neq=nn
 ALLOCATE(ell(nels),num(nod),prop(nprops,np_types),etype(nels),           &
   kp(nod,nod),g_num(nod,nels),kdiag(neq),loads(0:neq),disps(0:neq))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)ell; READ(10,*)g_num; kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   num=g_num(:,iel); CALL fkdiag(kdiag,num)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq))); kv=zero
!-----------------------global conductivity matrix assembly---------------
 elements_2: DO iel=1,nels
   CALL rod_km(kp,prop(1,etype(iel)),ell(iel))
   num=g_num(:,iel); CALL fsparv(kv,kp,num,kdiag)
 END DO elements_2; kvh=kv
!-----------------------specify boundary values---------------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   kv(kdiag(node))=kv(kdiag(node))+penalty
   loads(node)=kv(kdiag(node))*value
 END IF
!-----------------------equation solution--------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
!-----------------------retrieve nodal net flow rates---------------------
 CALL linmul_sky(kvh,loads,disps,kdiag)
 WRITE(11,'(/A)')"  Node Total Head  Flow rate"; disps(0)=zero
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
 WRITE(11,'(/A)')"       Inflow      Outflow"
 WRITE(11,'(5X,2E12.4)')                                                 &
   SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero)
STOP
END PROGRAM p71
```

Scalar integers:

`fixed_freedoms`	number of fixed total heads
`i`	simple counter
`iel`	simple counter
`iwp`	`SELECTED_REAL_KIND(15)`
`k`	node number
`loaded_nodes`	number of fixed source/sink nodes
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nn`	number of nodes in the mesh
`nod`	number of nodes per element
`nprops`	number of material properties
`np_types`	number of different property types

Scalar reals:

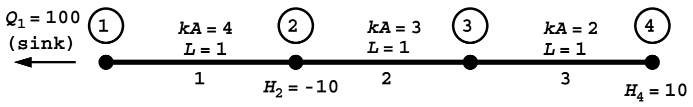
`penalty`	set to 1×10^{20}
`zero`	set to 0.0

Dynamic integer arrays:

`etype`	element property types vector
`g_num`	global element node numbers matrix
`kdiag`	diagonal term location vector
`node`	fixed nodes vector
`num`	element node numbers vector

Dynamic real arrays:

`disps`	net nodal inflow/outflow
`ell`	element lengths
`kc`	element conductivity matrix
`kv`	global conductivity matrix
`kvh`	copy of `kv`
`loads`	nodal total heads
`prop`	element properties
`value`	fixed values of total heads



```
nels   nn   np_types
3       4     3

prop(ka)
4.0  3.0  2.0

etype
1 2 3

ell
1.0  1.0  1.0

g_num
1 2  2 3  3 4

loaded_nodes,(k,loads(k),i=1,loaded_nodes)
1
1  -100.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
2
2  -10.0   4  10.0
```

Figure 7.1 Mesh and data for first Program 7.1 example

Figure 7.1 shows a string of three elements attached end to end. Each element has the same length, but different permeability properties. In this context, the property applied to each element is *kA* (analogous to *EA* in Program 4.1), namely the product of the permeability and the cross-sectional area of each element. A fixed steady outflow or "sink" of −100.0 (negative sign denotes outflow) is applied at node 1, and the total head is fixed to −10.0 and 10.0 at nodes 2 and 4 respectively. The data involves reading the number of elements `nels`, the number of nodes `nn`, and the number of property types `np_types`. In this case there are 3 property types, one for each element, so with `np_types>1` the `etype` data is read next, indicating that element 1 has a *kA* of 4.0, element 2 has a *kA* of 3.0, and so on. The element lengths `ell` are read, followed by the node numbers of each element `g_num`. In the case of a string of elements such as this, the `g_num` data is quite

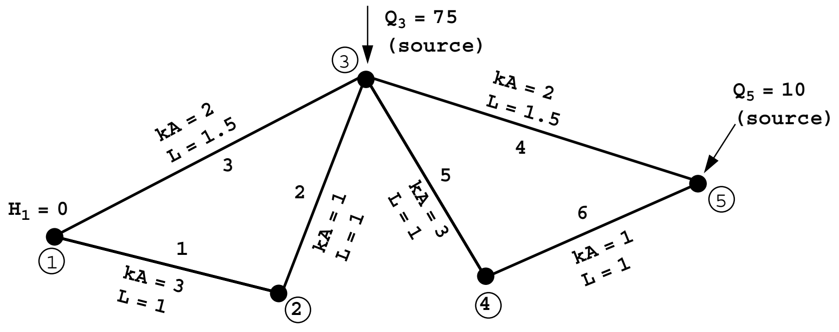
```
There are    4 equations and the skyline storage is    7

 Node Total Head    Flow rate
   1 -0.3500E+02   -0.1000E+03
   2 -0.1000E+02    0.7600E+02
   3 -0.2000E+01    0.3553E-14
   4  0.1000E+02    0.2400E+02

      Inflow        Outflow
     0.1000E+03    -0.1000E+03
```

Figure 7.2 Results from first Program 7.1 example

predictable. Finally the fixed source/sink values and fixed total head values are read through the `loaded_nodes` and `fixed_freedoms` data. The output shown in Figure 7.2 indicates the total head at nodes 1 and 3 to be −35.00 and −2.00 respectively, and the net flow rates at nodes 2 and 4 to be inflows of 76.0 and 24.0 respectively. The final line of output confirms the continuity condition that the total flow in, is the same as the total flow out.

The second example shown in Figure 7.3 is of a pipe network involving 6 elements and 5 nodes. There are three different property groups spread across the elements, which do not all have the same lengths. The `g_num` data gives the connectivity of the network. The boundary conditions include sources of 75.0 and 10.0 at nodes 3 and 5 respectively, and a total head at node 1 equal to zero. The output shown in Figure 7.4 indicates an outflow of 85.0 at node 1, and total heads at nodes 2, 3, 4, and 5 of 10.2, 40.8, 42.0, and 45.6 respectively.

It should be noted that in problems of this type, all nodal boundary conditions are fixed to either a net flow rate or a total head. If the data prescribes both the net flow rate and the total head at a particular node, the total head takes priority as the dependent variable.



```
nels   nn   np_types
6       5     3

prop(ka)
3.0  2.0  1.0

etype
1  3  2  2  1  3

ell
1.0  1.0  1.5  1.5  1.0  1.0

g_num
1 2  2 3  1 3  3 5  3 4  4 5

loaded_nodes,(k,loads(k),i=1,loaded_nodes)
2
3 75.0   5  10.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
1
1  0.0
```

Figure 7.3 Mesh and data for second Program 7.1 example

```
 There are    5 equations and the skyline storage is   11

  Node Total Head   Flow rate
     1  0.8500E-18  -0.8500E+02
     2  0.1020E+02   0.0000E+00
     3  0.4080E+02   0.7500E+02
     4  0.4200E+02  -0.2842E-13
     5  0.4560E+02   0.1000E+02

        Inflow       Outflow
        0.8500E+02  -0.8500E+02
```

Figure 7.4 Results from second Program 7.1 example

The default net flow rate at all nodes is set initially to zero, so only non-zero values need to be input as data.

Program 7.2 Plane or axisymmetric analysis of steady seepage using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$- or $y(z)$- direction.

```
PROGRAM p72
!-------------------------------------------------------------------------
! Program 7.2 Plane or axisymmetric analysis of steady seepage using
!             4-node rectangular quadrilaterals. Mesh numbered
!             in x(r)- or y(z)- direction.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,nci,ndim=2,nels,neq,nip=4,  &
   nod=4,nn,np_types,nxe,nye
 REAL(iwp)::det,one=1.0_iwp,penalty=1.0e20_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::dir,element='quadrilateral',type_2d
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),kdiag(:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),der(:,:),deriv(:,:),disps(:),fun(:),   &
   gc(:),g_coord(:,:),jac(:,:),kay(:,:),kp(:,:),kv(:),kvh(:),loads(:),    &
   points(:,:),prop(:,:),value(:),weights(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)type_2d,dir,nxe,nye,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
 ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),              &
   jac(ndim,ndim),weights(nip),der(ndim,nod),deriv(ndim,nod),             &
   kp(nod,nod),num(nod),g_num(nod,nels),kay(ndim,ndim),etype(nels),       &
   x_coords(nxe+1),y_coords(nye+1),prop(ndim,np_types),gc(ndim),fun(nod), &
   kdiag(neq),loads(0:neq),disps(0:neq))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
   CALL fkdiag(kdiag,num)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq))); WRITE(11,'(2(A,I5))')         &
   "There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); kv=zero; gc=one
!-----------------------global conductivity matrix assembly---------------
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kp=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); IF(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
     kp=kp+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
   END DO gauss_pts_1; CALL fsparv(kv,kp,num,kdiag)
 END DO elements_2; kvh=kv
!-----------------------specify boundary values---------------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   kv(kdiag(node))=kv(kdiag(node))+penalty
   loads(node)=kv(kdiag(node))*value
 END IF
!-----------------------equation solution---------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
!-----------------------retrieve nodal net flow rates---------------------
 CALL linmul_sky(kvh,loads,disps,kdiag)
 WRITE(11,'(/A)')"  Node Total Head  Flow rate"
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
 disps(0)=zero; WRITE(11,'(/A)')"       Inflow      Outflow"
 WRITE(11,'(5X,2E12.4)')                                                &
   SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero)
 READ(10,*)nci; IF(nod==4)CALL contour(loads,g_coord,g_num,nci,13)
STOP
END PROGRAM p72
```

New scalar integers:

`nci`	number of contour intervals required
`ndim`	number of dimensions
`nip`	number of integrating points
`nxe`	number of elements in the $x(r)$-direction
`nye`	number of elements in the $y(r)$-direction

New scalar reals:

`det`	determinant of the Jacobian matrix
`one`	set to 1.0

Scalar characters:

`dir`	element and node numbering direction
`element`	element type
`type_2d`	type of 2D analysis

New dynamic real arrays:

`coord`	element nodal coordinates
`der`	shape function derivatives with respect to local coordinates

`deriv`	shape function derivatives with respect to global coordinates
`fun`	shape functions
`gc`	integrating point coordinates
`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`kay`	permeability matrix
`points`	integrating point local coordinates
`weights`	weighting coefficients
`x_coords`	$x(r)$-coordinates of mesh layout
`y_coords`	$y(z)$-coordinates of mesh layout

This program is for the analysis of 2D steady seepage problems under plane or axisymmetric conditions, and is analogous to Program 5.1 in Chapter 5. In order to simplify the data however, the examples presented here use 4-node rectangular elements only (`element='quadrilateral'` and `nod=4`). The program includes graphics subroutines `mesh` and `contour` which generate PostScript files containing, respectively, images of the finite element mesh (held in `fe95.msh`), and a contour map of the dependent variable (held in the `fe95.con`). Contouring is currently restricted to meshes made up of 4-node quadrilateral elements.

The first example in Figure 7.5 shows a typical problem of steady seepage beneath an impermeable sheet pile wall. The total head loss across the wall has been normalised to 100 units, but due to symmetry only half the problem needs to be analysed, with a total head loss of 50 units. Figure 7.6 shows the mesh and data that will be used to analyse the problem.

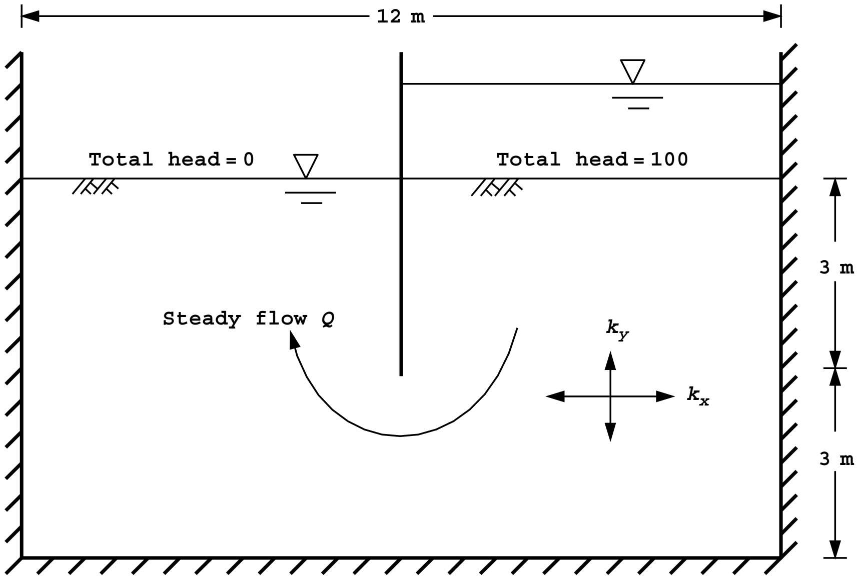
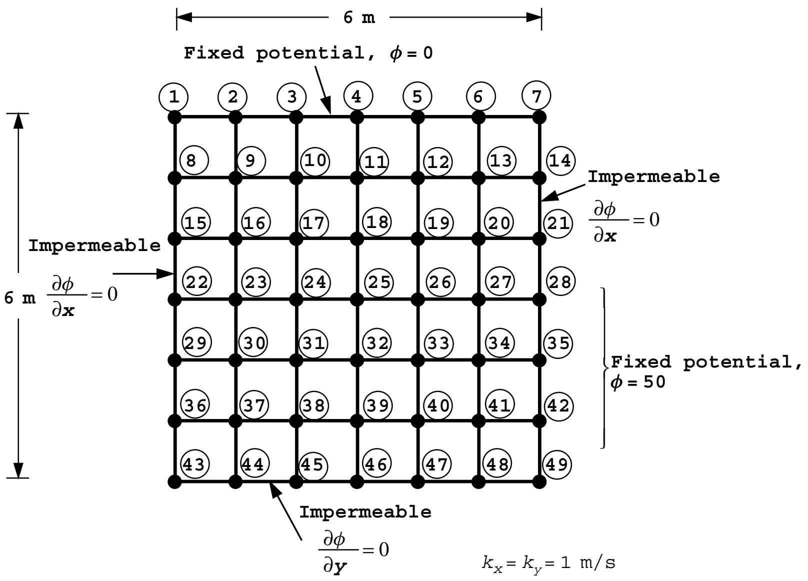


Figure 7.5 Steady flow under a single sheet pile


6 m
Fixed potential, $\phi = 0$
Impermeable
$\frac{\partial \phi}{\partial x} = 0$
Impermeable
$\frac{\partial \phi}{\partial x} = 0$
6 m
Fixed potential, $\phi = 50$
Impermeable
$\frac{\partial \phi}{\partial y} = 0$
$k_x = k_y = 1$ m/s

```
type_2d  dir
'plane'  'x'

nxe  nye  np_types
6    6     1

prop(kx,ky)
1.0  1.0

etype(not needed)

x_coords, y_coords
0.0  1.0  2.0  3.0  4.0  5.0  6.0
0.0 -1.0 -2.0 -3.0 -4.0 -5.0 -6.0

loaded_nodes
0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
11
 1  0.0     2  0.0      3  0.0     4  0.0
 5  0.0     6  0.0      7  0.0
28 50.0    35 50.0     42 50.0    49 50.0

nci
10
```

Figure 7.6 Mesh and data for first Program 7.2 example

The first line of data reads `type_2d` and `dir` indicating that a plane analysis is to be performed with element and node numbering in the x-direction. The second line indicates that the rectangular mesh consists of six columns (`nxe`) and six rows (`nye`) of elements, and that there is only one property type (`np_types`) in this homogeneous example. The third line reads the x- and y-direction permeabilities, k_x and k_y into the property array `prop`, and since there is only one property type in this problem, the `etype` data is not required. The fourth and fifth lines give, respectively, the x- (`x_coords`) and y-coordinates (`y_coords`) of the lines that make up the mesh. The zero `loaded_nodes` on the sixth line indicates that no internal sources or sinks are applied in this case. The seventh to tenth lines indicate that there are 11 fixed total head values (`fixed_freedoms`) values at the up- and downstream boundaries of the mesh. The last line of data indicates that after the total head values have been computed, a contour map will be produced with 10 (`nci`) contour intervals (or equipotential drops n_d, see equation (7.2)).

The fixed potential boundary conditions (equal to either zero or 50) are fixed through the data as described above. All other boundaries are "no-flow" or impermeable, so a boundary condition of $\partial\phi/\partial n = 0$ is required, which is obtained by default at the boundaries of the mesh by taking no further action.

The program assumes a rectangular mesh made up of rectangular elements, with nodal coordinates and connectivity generated by the library subroutine `geom_rect`.

After scanning the elements to determine the storage requirements, the program uses numerical integration to form the element conductivity matrices `kc`, which are then assembled into a global conductivity matrix `kv`. The sequence of operations described by the `elements_2` loop bears a striking similarity to the integration of an element stiffness matrix used, for example, in Program 5.1. Program 7.2 is actually simpler, because the derivative array `deriv` is used directly in the products described by (3.63).

Following the solution of the "equilibrium" equations, which is performed by library subroutines `sparin` and `spabac`, the nodal potentials are held in the vector `loads` and printed. In order to retrieve the nodal flow rates `disps`, the matrix `kvh`, which is a copy of the global conductivity matrix `kv`, is multiplied by the nodal potentials `loads` by library subroutine `linmul_sky`. Examination of `disps` reveals that the majority of net flow rates corresponding to internal nodes are zero, the only non-zero values occurring at the boundary nodes that had their total head values fixed. If we had chosen to include an internal source or sink as data using `loaded_nodes`, this would have appeared at the appropriate node in the `disps` vector.

Finally, the net inflow and outflow through the system is computed by summing, respectively, the positive and negative terms in `disps`. The output from Program 7.2 is shown in Figure 7.7. As expected the inflow and outflow values are identical and give a steady state flow rate of 48.6. The Method of Fragments for this constrained seepage problem (e.g. Griffiths, 1984) would predict a flow rate of around 47. The theoretical solution for a sheet pile wall embedded to half the depth of a stratum of similar soil in a domain which extends to infinity laterally would be exactly 50.0.

A good way to visualise the results of a seepage analysis such as this is to draw a contour map of the nodal potentials. Figure 7.8 shows a contour map of the total heads *and* the stream functions that would be computed using a rather more refined mesh (50×50 elements) than that shown in Figure 7.6. Both sides of the wall are shown for clarity, although only half the problem was actually analysed. The stream function problem has not been solved in this example, however it could easily be included by solving the 'inverse' problem given by:

$$\frac{1}{k_y}\frac{\partial^2\psi}{\mathrm{d}x^2} + \frac{1}{k_x}\frac{\partial^2\psi}{\mathrm{d}y^2} = 0 \tag{7.1}$$

where ψ is the stream function. The boundary conditions for the stream problem must now be "inverted", thus those boundaries that had fixed values in the potential problem such as the up- and downstream boundaries, have $\partial\psi/\partial n = 0$ boundary conditions in the

```
There are   49 equations and the skyline storage is  385

  Node Total Head     Flow rate
    1  0.2926E-19   -0.2926E+01
    2  0.6063E-19   -0.6063E+01
    3  0.6708E-19   -0.6708E+01
    4  0.7810E-19   -0.7810E+01
    5  0.9184E-19   -0.9184E+01
    6  0.1042E-18   -0.1042E+02
    7  0.5453E-19   -0.5453E+01
    8  0.5716E+01   -0.8882E-15
    9  0.5921E+01   -0.1776E-14
   10  0.6553E+01   -0.0000E+00
.
.
.
   40  0.3378E+02    0.1066E-13
   41  0.4130E+02    0.0000E+00
   42  0.5000E+02    0.9201E+01
   43  0.2196E+02   -0.2220E-14
   44  0.2272E+02   -0.1776E-14
   45  0.2502E+02    0.7105E-14
   46  0.2897E+02    0.1510E-13
   47  0.3464E+02    0.0000E+00
   48  0.4185E+02   -0.1776E-13
   49  0.5000E+02    0.4260E+01

       Inflow        Outflow
       0.4857E+02   -0.4857E+02
```

Figure 7.7 Results from first Program 7.2 example

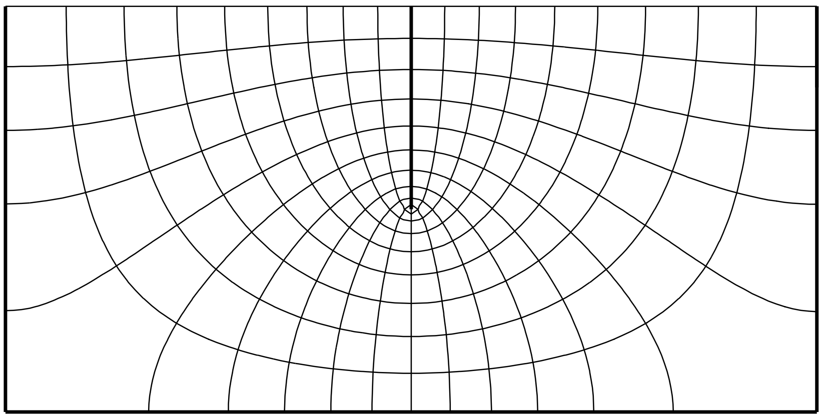


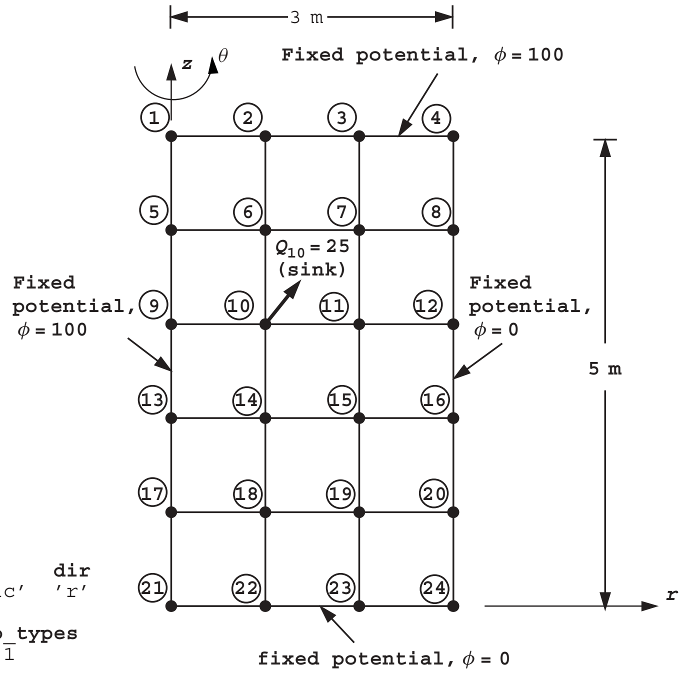
Figure 7.8 Flow net of seepage beneath a sheet pile wall from first Program 7.2 example

stream problem, and boundaries that had $\partial\phi/\partial n = 0$ conditions in the potential problem, such as at impermeable boundaries, are given fixed values of the stream function. In order to choose a contour interval which satisfies the usual flow-net rules involving “square” regions, it is suggested that when solving the stream problem, the uppermost streamline (in this case the wall) is fixed equal to the flow rate ($\psi = 48.6$), and the lowest streamline (the impermeably boundary) is fixed to zero ($\psi = 0$). The required number of flow channels

n_f for the stream contour map, can then be computed from,

$$n_f = \frac{Q\, n_d}{k\, H} \tag{7.2}$$

where Q is the flow rate computed from the potential problem, k is the (isotropic) permeability of the soil, n_d is the number of equipotential drops chosen for the total head contour



```
type_2d          dir
'axisymmetric'   'r'

nxe  nye  np_types
3     5     1

prop(kx,ky)
1.0  1.0

etype(not needed)

x_coords, y_coords
0.0  1.0  2.0  3.0
0.0 -1.0 -2.0 -3.0 -4.0 -5.0

loaded_nodes,(k,loads(k),i=1,loaded_nodes)
1
10  -25.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
 16
 1 100.0   2 100.0  3 100.0   4   0.0     5 100.0   8   0.0   9 100.0
12   0.0  13 100.0 16   0.0  17 100.0     20  0.0  21   0.0  22   0.0
23   0.0  24   0.0
20
```

Figure 7.9 Mesh and data for second Program 7.2 example

plot, and H is the total head loss between the up- and downstream boundaries. Finally, n_f should be rounded to the nearest whole number and input to the stream function analysis as the number of stream contour intervals `nci`.

A second example of the use of Program 7.2 is shown in Figure 7.9, and represents a radial plane of a cylinder of porous material. The model subtends one radian at the axis of rotational symmetry. The boundary conditions consist of a fixed total head of 100 units on the top of the cylinder and on the central axis ($r = 0$). The outer surface of the cylinder and the bottom surface are fixed to zero. The data are very similar to those of the previous example, but with `type_2d` set to `'axisymmetric'` and `dir` set to `'r'` because the node and element numbering is now in the radial direction. Although axisymmetry adds an extra order of r to the terms to be integrated, the "mass" and conductivity matrices of rectangular 4-node elements in radial planes, are still exactly integrated with `nip=4`. When performing axisymmetric analysis with non-rectangular quadrilateral elements however, higher orders of numerical integration should be investigated, and slightly different results can be expected as `nip` is increased. Customised quadrature rules for axisymmetric analysis are available (see e.g. Griffiths, 1991).

In this example, a steady point sink of -25.0 m^3/s/radian is applied to node 10. The computed results are shown in Figure 7.10. In addition to the usual flow rates recorded at the boundary nodes, the fluid removed from the system at node 10 also appears in the "Flow rate" column as -25.0. The net inflow (outflow) from the entire system is computed to be 362.7 m^3/s/radian.

```
There are   24 equations and the skyline storage is  122

  Node Total Head     Flow rate
    1  0.1000E+03     0.6068E+01
    2  0.1000E+03     0.4558E+02
    3  0.1000E+03     0.1878E+03
    4  0.6925E-18    -0.6925E+02
    5  0.1000E+03     0.1726E+02
    6  0.6359E+02     0.3553E-14
    7  0.3310E+02    -0.4086E-13
    8  0.1255E-17    -0.1255E+03
    9  0.1000E+03     0.2863E+02
   10  0.3283E+02    -0.2500E+02
.
.
.
   19  0.6183E+01     0.7105E-14
   20  0.1421E-18    -0.1421E+02
   21  0.3278E-19    -0.3278E+01
   22  0.2631E-18    -0.2631E+02
   23  0.1396E-18    -0.1396E+02
   24  0.5152E-19    -0.5152E+01

       Inflow         Outflow
       0.3627E+03    -0.3627E+03
```

Figure 7.10 Results from second Program 7.2 example

Program 7.3 Analysis of plane free-surface flow using 4-node quadrilaterals. "Analytical" form of element conductivity matrix.

```
PROGRAM p73
!-------------------------------------------------------------------------
! Program 7.3 Analysis of plane free-surface flow using 4-node
!             quadrilaterals. "Analytical" form of element conductivity
!             matrix.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_down,fixed_seep,fixed_up,i,iel,iters,k,limit,nci,ndim=2,  &
   nels,neq,nod=4,nn,nxe,nye,np_types
 REAL(iwp)::d180=180.0_iwp,hdown,hup,initial_height,one=1.0_iwp,          &
   penalty=1.e20_iwp,tol,zero=0.0_iwp; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),kdiag(:),node_down(:),          &
   node_seep(:),node_up(:),num(:)
 REAL(iwp),ALLOCATABLE::angs(:),bottom_width(:),coord(:,:),disps(:),      &
   g_coord(:,:),kay(:,:),kp(:,:),kv(:),kvh(:),loads(:),oldpot(:),         &
   prop(:,:),surf(:),top_width(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,tol,limit,np_types; nels=nxe*nye
 nn=(nxe+1)*(nels/nxe+1); neq=nn
 ALLOCATE(g_coord(ndim,nn),coord(nod,ndim),bottom_width(nxe+1),           &
   top_width(nxe+1),surf(nxe+1),angs(nxe+1),kp(nod,nod),num(nod),         &
   g_num(nod,nels),prop(ndim,np_types),kdiag(neq),kay(ndim,ndim),         &
   etype(nels),loads(0:neq),disps(0:neq),oldpot(0:neq))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)bottom_width; READ(10,*)top_width; READ(10,*)initial_height
 surf=initial_height
 angs=ATAN(surf/(top_width-bottom_width))*d180/acos(-one)
 READ(10,*)hup,fixed_up; ALLOCATE(node_up(fixed_up)); READ(10,*)node_up
 READ(10,*)hdown,fixed_down; ALLOCATE(node_down(fixed_down))
 READ(10,*)node_down; fixed_seep=nels/nxe-fixed_down
 ALLOCATE(node_seep(fixed_seep))
 DO i=1,fixed_seep; node_seep(i)=i*(nxe+1)+1; END DO; kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_freesurf(iel,nxe,fixed_seep,fixed_down,                      &
     hdown,bottom_width,angs,surf,coord,num); CALL fkdiag(kdiag,num)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   "There are ",neq,"  equations and the skyline storage is ",kdiag(neq)
!-----------------------global conductivity matrix assembly---------------
 oldpot=zero; iters=0
 its: DO
   iters=iters+1; kv=zero
   elements_2: DO iel=1,nels
     kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
     CALL geom_freesurf(iel,nxe,fixed_seep,fixed_down,                    &
       hdown,bottom_width,angs,surf,coord,num)
     g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
```

```
    CALL seep4(kp,coord,kay); CALL fsparv(kv,kp,num,kdiag)
   END DO elements_2; kvh=kv
!-----------------------specify boundary values--------------------------
   loads=zero; kv(kdiag(node_up))=kv(kdiag(node_up))+penalty
   loads(node_up)=kv(kdiag(node_up))*hup
   kv(kdiag(node_down))=kv(kdiag(node_down))+penalty
   loads(node_down)=kv(kdiag(node_down))*hdown
   kv(kdiag(node_seep))=kv(kdiag(node_seep))+penalty
   DO i=1,fixed_seep
     loads(node_seep(i))=kv(kdiag(node_seep(i)))*                       &
       (hdown+(surf(1)-hdown)*(fixed_seep+1-i)/(fixed_seep+1))
   END DO
!-----------------------equation solution--------------------------------
   CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
   surf(1:nxe)=loads(1:nxe)
!-----------------------check convergence--------------------------------
   CALL checon(loads,oldpot,tol,converged)
   IF(converged.OR.iters==limit)EXIT
 END DO its; CALL linmul_sky(kvh,loads,disps,kdiag)
 WRITE(11,'(/A)')"  Node Total Head  Flow rate"
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
 disps(0)=zero; WRITE(11,'(/A)')"       Inflow      Outflow"
 WRITE(11,'(5X,2E12.4)')                                                &
    SUM(disps,MASK=disps<zero),SUM(disps,MASK=disps>zero)
 WRITE(11,'(/A,I3,A)')" Converged in",iters," iterations"
 CALL mesh(g_coord,g_num,12)
 READ(10,*)nci; CALL contour(loads,g_coord,g_num,nci,13)
STOP
END PROGRAM p73
```

New scalar integers:

`fixed_down`	number of nodes on downstream side
`fixed_seep`	number of nodes on seepage surface
`fixed_up`	number of nodes on upstream side
`iters`	counts free-surface iterations
`limit`	iteration ceiling

New scalar reals:

`hdown`	fixed total head on downstream side
`hup`	fixed total head on upstream side
`d180`	set to 180.0
`initial_height`	initial height of free surface to start process
`tol`	convergence tolerance

Scalar logical:

`converged`	set to `.TRUE.` if mesh has converged

New dynamic integer arrays:

`node_down`	nodes fixed on downstream side
`node_seep`	nodes fixed on downstream seepage surface
`node_up`	nodes fixed on upstream side

New dynamic real arrays:

`angs`	angles to horizontal made by sloping mesh lines
`bottom_width`	x-coordinates of nodes at base of mesh
`oldpot`	nodal total head values from previous iteration
`surf`	current total head values of free surface
`top_width`	x-coordinates of initial nodes at top of mesh

In this program we consider a boundary condition frequently met in geomechanics in relation to the flow of water through dams. Free-surface problems involve an upper boundary, the location of which is not known *a priori*, so an iterative procedure is required to find it. This iteration can be done in several ways; for example, a fixed mesh can be used and nodes separated into “active” and “inactive” ones depending upon whether fluid exists at that point. An alternative strategy is to use the present program, whereby the mesh is deformed so that its upper surface ultimately coincides with the free surface. A summary of the boundary conditions is given in Figure 7.11

The analysis starts by assuming an initial position for the free surface. Solution of Laplace’s equation gives values of the total head along the free-surface nodes which will not in general equal the elevation of the upper surface of the mesh. The elevations of the nodes along the upper surface are therefore adjusted to equal the total head values just calculated at those locations. In order to avoid distorted elements, the library geometry subroutine `geom_freesurf` ensures that the nodes beneath the top surface are evenly distributed. The geometry subroutine is designed for solving free surface problems with initially trapezoidal meshes and counts nodes and elements in the x-direction. The analysis is then repeated with the new mesh. Since many of the coordinates have changed, the conductivity matrices of all the elements must be re-computed and assembled into the global system. In order to avoid the need for numerical integration of the element conductivity matrices at each iteration, library subroutine `seep4` computes the element

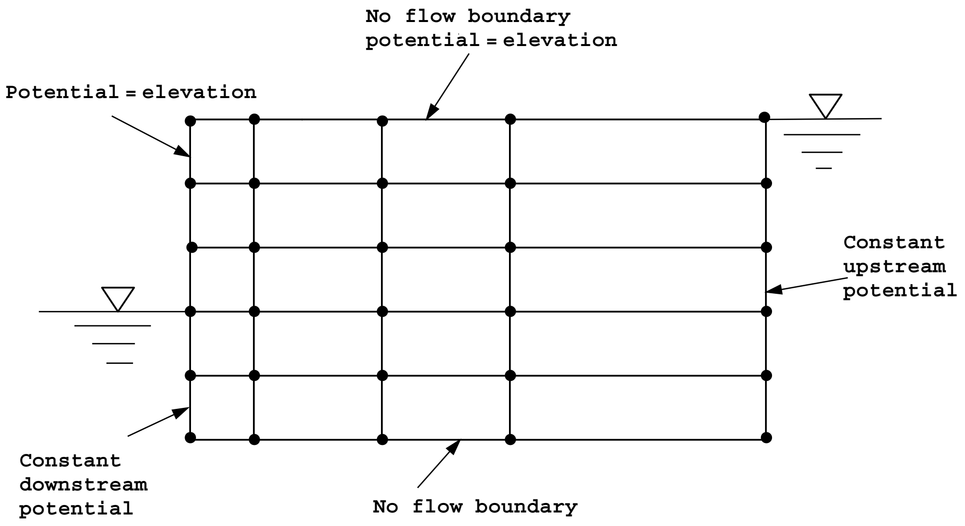


Figure 7.11 Boundary conditions for free surface flow

conductivity matrices kc "analytically" (see Section 3.2.2). The assembly is made into a global conductivity matrix kv stored as a skyline in the usual way. Solution of the modified problem leads to another set of nodal total head values and further updating of the mesh nodal coordinates. This process is repeated until the change in computed total head values from one iteration to the next is less than a tolerance value tol. The convergence check is performed by library subroutine checon, which outputs the logical variable converged. The subroutine sets converged to .TRUE. if the solution has converged and to .FALSE. if another iteration is required.

The first example shown in Figure 7.12 is of a vertical-sided dam. The free surface described by nodes 1 through 9 at the top of the mesh is initially assumed to be horizontal. The initial data relates to the number of elements in the x-(nxe) and y-(nxe) directions, and this is followed by the tolerance tol, set to 0.01 and the iteration ceiling limit, set to 20. The dam is homogeneous in this example, so np_types is set to 1. The permeabilities in the x- and y-directions are read followed by the x-coordinates of the bottom bottom_width and top nodes top_width of the starting mesh. The next line of data reads the initial height of the horizontal free surface initial_height which in this example is set to 7.0. The value of the upstream total head value of hup=7.0 is then read, followed by the number of nodes to be fixed on the upstream side fixed_up and their node numbers node_up. Similarly on the downstream side, the total head is set to hdown=2.0 followed by fixed_down and node_down. The final data nci relates to the number of contour intervals to be plotted.

The output for this example is shown in Figure 7.13 and the Postscript output file fe95.msh shown in Figure 7.14 gives the deformed mesh at convergence. The graphics file fe95.con produces the contour map of total head values given in Figure 7.15.

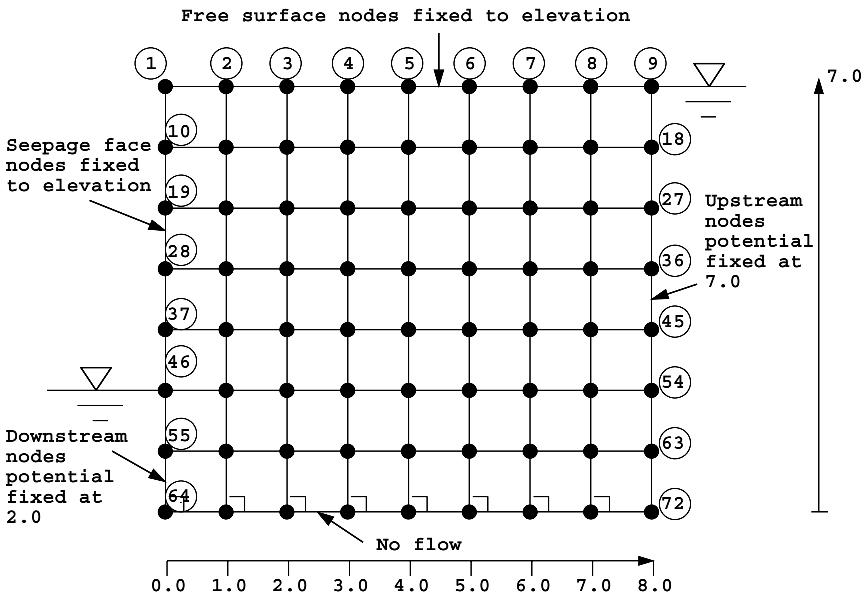

Free surface nodes fixed to elevation
1
2
3
4
5
6
7
8
9
10
18
19
27
28
36
37
45
46
54
55
63
64
72
Seepage face nodes fixed to elevation
Upstream nodes potential fixed at 7.0
Downstream nodes potential fixed at 2.0
No flow
7.0
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0

Figure 7.12 Mesh and data for first Program 7.3 example (*Continued on page 336*)

```
nxe  nye  tol  limit  np_types
8     7   0.01  20      1

prop(kx,ky)
0.001  0.001

etype(not needed)

bottom_width, top_width
0.0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0
0.0   1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0

initial_height
7.0

hup,fixed_up,(node_up(i),i=fixed_up)
7.0
8
9  18  27  36  45  54  63  72

hdown,fixed_down,(node_down(i),i=fixed_down)
2.0
3
46  55  64

nci
20
```

Figure 7.12 (*Continued from page 335*)

```
There are    72  equations and the skyline storage is   703

  Node Total Head    Flow rate
    1  0.2759E+01    0.8674E-18
    2  0.3784E+01   -0.2602E-17
    3  0.4498E+01   -0.1301E-17
    4  0.5062E+01   -0.1301E-17
    5  0.5571E+01    0.0000E+00
    6  0.6016E+01    0.1301E-17
    7  0.6409E+01    0.3036E-17
    8  0.6746E+01   -0.2602E-17
    9  0.7000E+01    0.1105E-03
   10  0.2645E+01   -0.3100E-04
.
.
.
   65  0.2840E+01   -0.4337E-18
   66  0.3619E+01   -0.4337E-18
   67  0.4310E+01    0.1409E-17
   68  0.4925E+01   -0.7589E-18
   69  0.5485E+01    0.1518E-17
   70  0.6009E+01   -0.2819E-17
   71  0.6510E+01   -0.3469E-17
   72  0.7000E+01    0.2432E-03

       Inflow         Outflow
     -0.2813E-02    0.2813E-02

 Converged in 10 iterations
```

Figure 7.13 Results from first Program 7.3 example

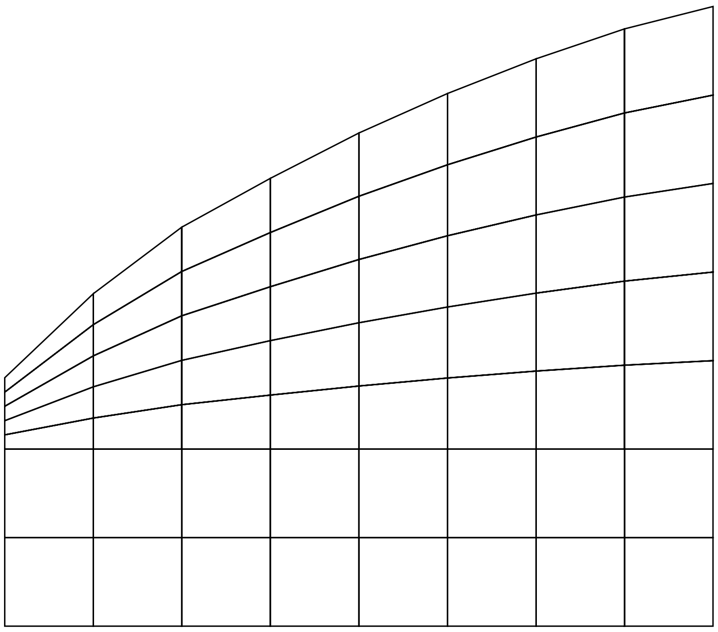


Figure 7.14 Computed free surface at convergence in vertical face dam analysis

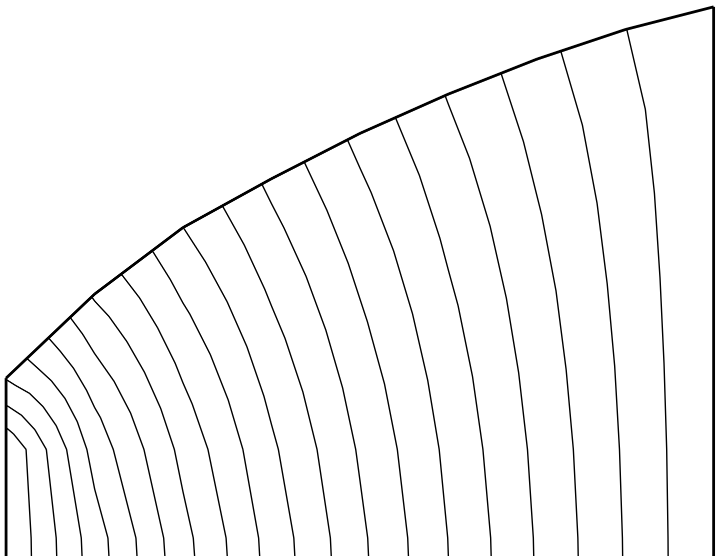


Figure 7.15 Equipotentials at convergence in vertical face dam analysis

The case of free-surface flow through a dam with vertical faces is a classical problem for which the Dupuit formula (see e.g. Verruijt, 1970) predicts a flow rate given by:

$$Q = \frac{k(H_1^2 - H_2^2)}{2D} \tag{7.3}$$

where $H_1 = 7.0$ m and $H_2 = 2.0$ m refer to the up- and downstream water elevations, $k =$ 0.001 m/s refers to the permeability (assumed isotropic and homogeneous) and $D = 8.0$ m refers to the width of the dam. The formula gives a flow rate of 0.00281 m^3/s/m which agrees exactly with the computed value in this case.

A second example of an earth dam with sloping sides and a relatively impermeable clay core is presented in Figure 7.16 with data in Figure 7.17. The initial mesh is trapezoidal, and starts with a horizontal free surface set at an elevation of 37.5 m which is also the height of the starting mesh. The nodes on the upstream face of the dam are also set at a total head of 37.5 m, while the bottom two nodes on the downstream side are fixed at a total head of 7.5 m. The initial mesh is defined by the x-coordinates of the nodes at the base and the top. There are two property types in this example, so the `etype` data is needed to allocate properties to the elements in the mesh. The middle 10 "columns" of elements represent the clay core.

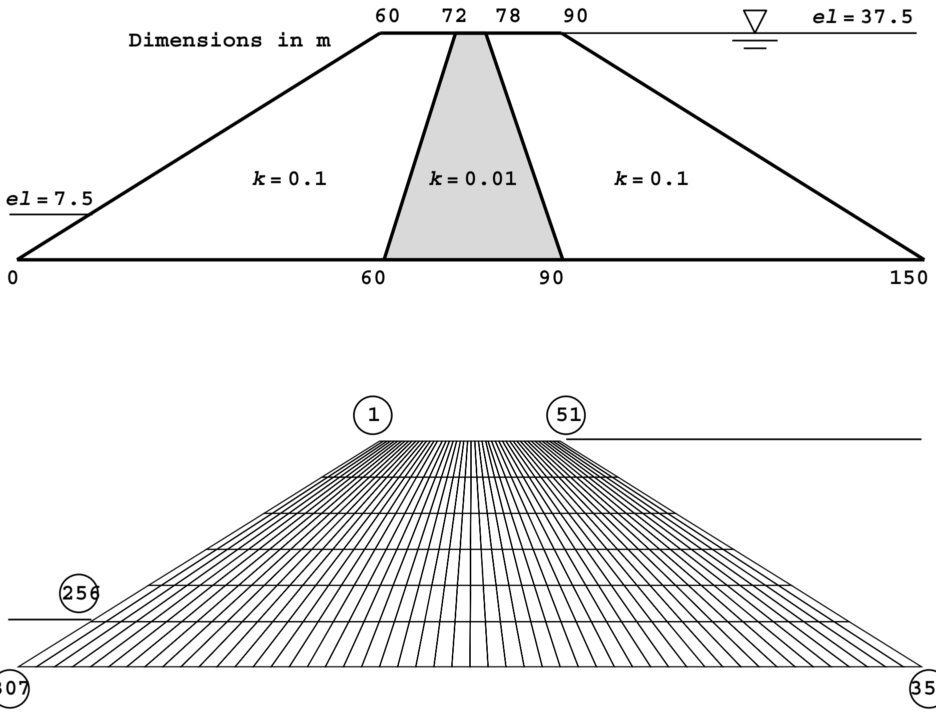


Figure 7.16 Configuration and mesh for embankment free surface analysis. Second Program 7.3 example

```
nxe  nye  tol  limit  np_types
50   6  0.01  20      2

prop(kx,ky)
0.1   0.1
0.01  0.01

etype
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
.
. etype data for elements 51-250 omitted
.
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

bottom_width, top_width
  0.0    3.0    6.0   9.0    12.0  15.0  18.0
 21.0   24.0   27.0  30.0   33.0  36.0  39.0
 42.0   45.0   48.0  51.0   54.0  57.0  60.0
 63.0   66.0   69.0  72.0   75.0  78.0  81.0
 84.0   87.0   90.0  93.0   96.0  99.0 102.0
105.0 108.0 111.0 114.0 117.0 120.0 123.0
126.0 129.0 132.0 135.0 138.0 141.0 144.0 147.0 150.0
 60.0   60.6   61.2  61.8   62.4  63.0  63.6
 64.2   64.8   65.4  66.0   66.6  67.2  67.8
 68.4   69.0   69.6  70.2   70.8  71.4  72.0
 72.6   73.2   73.8  74.4   75.0  75.6  76.2
 76.8   77.4   78.0  78.6   79.2  79.8  80.4
 81.0   81.6   82.2  82.8   83.4  84.0  84.6
 85.2   85.8   86.4  87.0   87.6  88.2  88.8  89.4  90.0

initial_height
 37.5

hup,(fixed_up,node_up(i),i=fixed_up)
 37.5
  7
 51  102  153  204  255  306  357

hdown,(fixed_down,node_down(i),i=fixed_down)
  7.5
  2
256  307

nci
 20
```

Figure 7.17 Data for second Program 7.3 example

During the mesh iterations, subroutine `geom_freesurf` ensures that the nodes are constrained to remain on the sloping lines and maintain even spacing in the y-direction. The output from this example is shown in Figure 7.18 indicating a steady flow of 0.3335 m^3/s/m. The deformed mesh, which took 11 iterations to converge, is shown in Figure 7.19 indicating how the free surface falls rapidly within the clay core.

```
         There are   357  equations and the skyline storage is 16313

           Node Total Head     Flow rate
              1  0.9407E+01     0.8882E-15
              2  0.9959E+01     0.0000E+00
              3  0.1081E+02     0.0000E+00
              4  0.1160E+02     0.8882E-15
              5  0.1229E+02     0.4441E-15
              6  0.1290E+02    -0.8882E-15
              7  0.1345E+02     0.1332E-14
              8  0.1396E+02    -0.8882E-15
              9  0.1443E+02     0.6661E-15
             10  0.1488E+02    -0.2220E-15
         .
         .
         .
            350  0.3743E+02     0.1776E-14
            351  0.3746E+02     0.3553E-14
            352  0.3747E+02    -0.8882E-14
            353  0.3749E+02    -0.7105E-14
            354  0.3749E+02    -0.1776E-14
            355  0.3750E+02     0.3553E-14
            356  0.3750E+02    -0.1776E-14
            357  0.3750E+02     0.4641E-03

                 Inflow         Outflow
                -0.3335E+00     0.3335E+00

          Converged in 11 iterations
```

Figure 7.18 Results from second Program 7.3 example

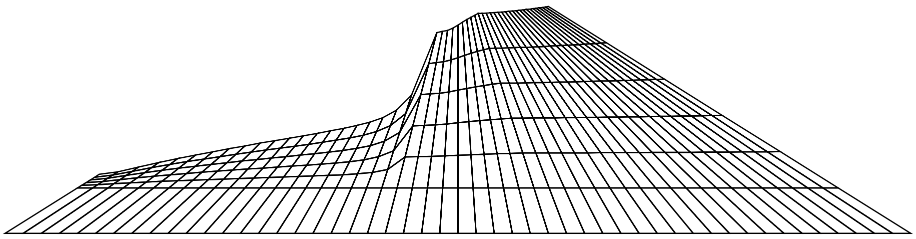


Figure 7.19 Computed free surface at convergence in embankment analysis

Program 7.4 General two- (plane) or three-dimensional analysis of steady seepage.

```
PROGRAM p74
!-------------------------------------------------------------------------
! Program 7.4 General two- (plane) or three-dimensional analysis of steady
!             seepage.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,nci,ndim,nels,neq,nip,nod,  &
   nn,np_types; CHARACTER(LEN=15)::element
 REAL(iwp)::det,penalty=1.0e20_iwp,zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),kdiag(:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),der(:,:),deriv(:,:),disps(:),          &
   g_coord(:,:),jac(:,:),kay(:,:),kp(:,:),kv(:),kvh(:),loads(:),          &
   points(:,:),prop(:,:),value(:),weights(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)element,nod,nels,nn,nip,ndim,np_types; neq=nn
 ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),etype(nels),  &
   jac(ndim,ndim),weights(nip),num(nod),g_num(nod,nels),der(ndim,nod),    &
   deriv(ndim,nod),kp(nod,nod),kay(ndim,ndim),prop(ndim,np_types),        &
   kdiag(neq),loads(0:neq),disps(0:neq))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)g_coord; READ(10,*)g_num
 IF(ndim==2)CALL mesh(g_coord,g_num,12); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel =1,nels
   num=g_num(:,iel); CALL fkdiag(kdiag,num)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 WRITE(11,'(2(A,I5))')                                                    &
  " There are",neq," equations and the skyline storage is",kdiag(neq)
 ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq))); kv=zero
 CALL sample(element,points,weights)
!-----------------------global conductivity matrix assembly---------------
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kp=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i); jac=MATMUL(der,coord)
     det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
     kp=kp+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
   END DO gauss_pts_1; CALL fsparv(kv,kp,num,kdiag)
 END DO elements_2; kvh=kv
!-----------------------specify boundary values---------------------------
 loads=zero; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   kv(kdiag(node))=kv(kdiag(node))+penalty
   loads(node)=kv(kdiag(node))*value
 END IF
!-----------------------equation solution---------------------------------
 CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
!-----------------------retrieve nodal net flow rates---------------------
 CALL linmul_sky(kvh,loads,disps,kdiag)
 WRITE(11,'(/A)')"  Node Total Head  Flow rate"
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
 disps(0)=zero; WRITE(11,'(/A)')"       Inflow      Outflow"
 WRITE(11,'(5X,2E12.4)')                                                  &
   SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero)
 IF(ndim==2.AND.nod==4)THEN
   READ(10,*)nci; CALL contour(loads,g_coord,g_num,nci,13)
 END IF
STOP
END PROGRAM p74
```

Program 7.4 can analyse steady confined seepage over any two- or three-dimensional domain with non-homogeneous and anisotropic material properties. The program is very similar to Program 5.4 for general elastic analysis of two- or three-dimensional solids, and can use any of the two- or three-dimensional elements referred to in this book (see Appendix B). The main difference from the programs described previously in this chapter, is that this program includes no "geometry" subroutine, so all nodal coordinates `g_coords` and element node numbers `g_num` must be provided as data. In addition, some of the variables that were previously fixed in the declaration statements, must now be read as data in order to identify the dimensionality of the problem and the type of element required. There are no variables required by this program that have not already been encountered in earlier programs of this chapter.

A three-dimensional seepage example is shown in Figure 7.20. The model represents one-eighth of a symmetrical cube with a point source of 100 units at its centroid with all outside faces maintained at a total head of zero. Referring to the figure, node numbers are indicated in circles and some of the element numbers have also been included. The example has 125 nodes and 64 elements.

The first line of data identifies the element type `element` which in this case is a '`hexahedron`', the number of nodes on each element `nod`, the number of elements `nels`, the number of nodes in the mesh `nn`, the number of integrating points `nip`, the number of dimensions of the problem `ndim`, and the number of property types `np_types`. It may be noted that numerical integration of an 8-node hexahedral element usually requires 8 Gauss points, (2 in each of the three coordinate directions), so `nip` is read as 8.

The problem includes 2 property types, so the next two lines of data provide the property values for each of the `np_types` groups. A 3D problem (`ndim=3`) such as this, requires 3 permeabilities terms (k_x, k_y and k_z) for each property group. In this example, the first group is applied to elements 1 to 32, which are isotropic with $k_x = k_y = k_z = 2$, and the

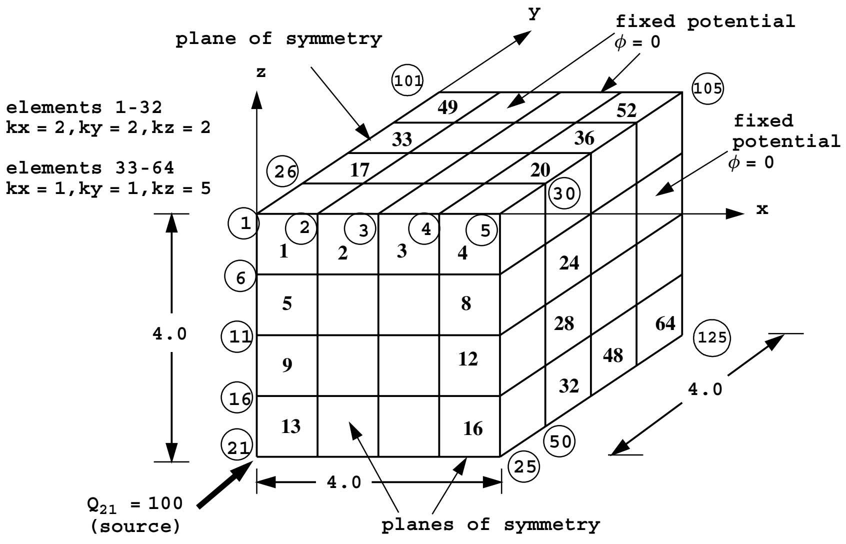


Figure 7.20 Mesh and data for Program 7.4 example (*Continued on page 343*)

```
element        nod  nels  nn   nip  ndim  np_types
'hexahedron'    8    64   125   8     3      2

prop(kx,ky,kz)
2.0  2.0  2.0     1.0  1.0  5.0

etype
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

g_coord
0.0  0.0  0.0     1.0  0.0  0.0     2.0  0.0  0.0     3.0  0.0  0.0
.
g_coord data for nodes 5-120 omitted here
.
0.0  4.0 -4.0     1.0  4.0 -4.0     2.0  4.0 -4.0     3.0  4.0 -4.0
4.0  4.0 -4.0

g_num
  6   1   2   7  31  26  27  32      7   2   3   8  32  27  28  33
.
g_num data for elements 3-62 omitted here
.
 98  93  94  99 123 118 119 124     99  94  95 100 124 119 120 125

loaded_nodes,(k,loads(k),i=1,loaded_nodes)
1      21  100.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
61
  1 0.0    2 0.0    3 0.0    4 0.0    5 0.0   10 0.0   15 0.0   20 0.0
.
fixed freedom data for 48 nodes omitted here
.
121 0.0  122 0.0  123 0.0  124 0.0  125 0.0
```

Figure 7.20 (*Continued from page 342*)

second group to elements 33 to 64, which are anisotropic with $k_x = k_y = 1$ and $k_z = 5$. The `etype` vector then reads the information required to match elements with property groups.

In this chapter we always assume that the principal axes of the permeability tensor coincide with the Cartesian coordinate axes leading to a diagonal property array `kay`. If this is not the case, the `kay` matrix will be fully populated with off-diagonal terms.

The next data involves the x,y,z coordinates of the `nn` nodes in the mesh read into `g_coord`, followed by the node numbers of each of the `nels` elements read into `g_num`. If dealing with a three-dimensional 8-node element for example, the order in which the node numbers are read must follow the sequence described for that element in Appendix B. Due to the volume of data required in this example, only a few lines of the `g_coord` and `g_num` data are actually shown in Figure 7.20.

There is one source at node 21 equal to 100.0 indicated in the `loaded_nodes` data. All the outside faces of the cube are fixed to zero, which requires 61 `fixed_freedoms` data.

If Program 7.4 is to be applied to a 2D analysis using 4-node quadrilateral elements, a final data input of the number of contour intervals `nci` is also required.

```
There are  125 equations and the skyline storage is 3225

 Node Total Head   Flow rate
   1  0.1543E-19  -0.1543E+01
   2  0.2732E-19  -0.2732E+01
   3  0.2028E-19  -0.2028E+01
   4  0.1032E-19  -0.1032E+01
   5  0.3514E-20  -0.3514E+00
   6  0.3319E+01  -0.1332E-14
   7  0.3183E+01  -0.7772E-15
   8  0.2152E+01   0.6661E-15
   9  0.1106E+01  -0.4441E-15
  10  0.1026E-19  -0.1026E+01
  11  0.9727E+01   0.3553E-14
  12  0.6905E+01   0.8882E-15
  13  0.5206E+01   0.4441E-15
  14  0.2141E+01   0.8882E-15
  15  0.2004E-19  -0.2004E+01
  16  0.1121E+02   0.3553E-14
  17  0.2098E+02   0.5995E-14
  18  0.6877E+01   0.4885E-14
  19  0.3154E+01  -0.1998E-14
  20  0.2684E-19  -0.2684E+01
  21  0.1689E+03   0.1000E+03
.
.
.
 120  0.2361E-20  -0.2361E+00
 121  0.5872E-20  -0.5872E+00
 122  0.8971E-20  -0.8971E+00
 123  0.7082E-20  -0.7082E+00
 124  0.3505E-20  -0.3505E+00
 125  0.1255E-20  -0.1255E+00

      Inflow       Outflow
      0.1000E+03  -0.1000E+03
```

Figure 7.21 Results from Program 7.4 example

A truncated version of the output from the program is shown in Figure 7.21. The total head is greatest at the central node, and equals about 169. Outflow occurs at all the outside nodes of the mesh where the total head was fixed to zero. For example, the outflow at node number 5 equals −0.3514.

Program 7.5 General two- (plane) or three-dimensional analysis of steady seepage. No global conductivity matrix assembly. Diagonally preconditioned conjugate gradient solver.

```
PROGRAM p75
!-------------------------------------------------------------------------
! Program 7.5 General two- (plane) or three-dimensional analysis of steady
!             seepage. No global conductivity matrix assembly.
!             Diagonally preconditioned conjugate gradient solver.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,k,loaded_nodes,nci,ndim, &
   nels,neq,nip,nod,nn,np_types
 REAL(iwp)::alpha,beta,cg_tol,det,one=1.0_iwp,penalty=1.0e20_iwp,up,      &
   zero=0.0_iwp
 CHARACTER(LEN=15)::element; LOGICAL::cg_converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),d(:),der(:,:),deriv(:,:),            &
   diag_precon(:),disps(:),g_coord(:,:),jac(:,:),kay(:,:),kp(:,:),      &
   loads(:),p(:),points(:,:),prop(:,:),store(:),storkp(:,:,:),u(:),     &
   value(:),weights(:),x(:),xnew(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)element,nod,nels,nn,nip,ndim,cg_tol,cg_limit,np_types; neq=nn
 WRITE(11,'(A,I5,A)')" There are",neq," equations"
 ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),etype(nels),  &
   jac(ndim,ndim),weights(nip),num(nod),g_num(nod,nels),der(ndim,nod),    &
   deriv(ndim,nod),kp(nod,nod),kay(ndim,ndim),prop(ndim,np_types),        &
   p(0:neq),loads(0:neq),x(0:neq),xnew(0:neq),u(0:neq),diag_precon(0:neq),&
   d(0:neq),disps(0:neq),storkp(nod,nod,nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)g_coord; READ(10,*)g_num; IF(ndim==2)CALL mesh(g_coord,g_num,12)
 diag_precon=zero; CALL sample(element,points,weights)
!----------element conductivity integration, storage and preconditioner---
 elements_1: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kp=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i); jac=MATMUL(der,coord)
     det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
     kp=kp+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
   END DO gauss_pts_1; storkp(:,:,iel)=kp
   DO k=1,nod; diag_precon(num(k))=diag_precon(num(k))+kp(k,k); END DO
 END DO elements_1
!-----------------------invert the preconditioner and get starting loads--
 loads=zero; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),                 &
     store(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   diag_precon(node)=diag_precon(node)+penalty
   loads(node)=diag_precon(node)*value; store=diag_precon(node)
 END IF
 diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
 d=diag_precon*loads; p=d; x=zero; cg_iters=0
 !-----------------------pcg equation solution-----------------------------
 pcg: DO
   cg_iters=cg_iters+1; u=zero
   elements_2: DO iel=1,nels
     num=g_num(:,iel); kp=storkp(:,:,iel); u(num)=u(num)+MATMUL(kp,p(num))
   END DO elements_2
   IF(fixed_freedoms/=0)u(node)=p(node)*store; up=DOT_PRODUCT(loads,d)
   alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha; loads=loads-u*alpha
   d=diag_precon*loads; beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
   CALL checon(xnew,x,cg_tol,cg_converged)
   IF(cg_converged.OR.cg_iters==cg_limit)EXIT
 END DO pcg
 WRITE(11,'(A,I5)')" Number of cg iterations to convergence was",cg_iters
!-----------------------retrieve nodal net flow rates---------------------
 loads=xnew; disps=zero
 elements_3: DO iel=1,nels
```

```
   num=g_num(:,iel); kp=storkp(:,:,iel)
   disps(num)=disps(num)+MATMUL(kp,loads(num))
 END DO elements_3; disps(0)=zero
 WRITE(11,'(/A)')"  Node Total Head  Flow rate"
 DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
 WRITE(11,'(/A)')"       Inflow      Outflow"
 WRITE(11,'(5X,2E12.4)')                                              &
 SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero)
 IF(ndim==2.AND.nod==4)THEN
   READ(10,*)nci; CALL contour(loads,g_coord,g_num,nci,13)
 END IF
STOP
END PROGRAM p75
```

New scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling

New scalar reals:

`alpha`	α from equations (3.22)
`beta`	β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\{\mathbf{R}\}_k$ from equations (3.22)

New scalar logical:

`cg_converged`	set to `.TRUE.` if pcg process has converged

New dynamic real arrays:

`d`	preconditioned rhs vector
`diag_precon`	diagonal preconditioner vector
`p`	"descent" vector used in equations (3.22)
`store`	stores augmented diagonal terms
`storkc`	holds element conductivity matrices
`u`	vector used in equations (3.22)
`x`	"old" solution vector
`xnew`	"new" solution vector

Program 7.5 is the final program in this chapter and is identical to Program 7.4 except for the equation solution strategy, which in this case uses the preconditioned conjugate gradient method with no global matrix assembly. The program is in many ways similar to Program 5.5 in Chapter 5 where the pcg method was first demonstrated.

All of the elements are looped in order to compute their conductivity matrices, which are stored in the array `storkc` for use later in the pcg solution algorithm. This loop called `elements_1` also builds the preconditioning matrix, which is simply the inverse of the diagonal terms in what would have been the assembled global conductivity matrix. The preconditioning matrix (stored as a vector) is called `diag_precon`. The section commented "pcg equation solution" carries out the vector operations described in equations (3.22) within the iterative loop labelled `pcg`. The matrix–vector multiply needed in the first of (3.22) is done using (3.23). The node number vector `num` "gathers" the appropriate components of `p`, to be multiplied by the element conductivity matrix `kc` retrieved from `storkc`. Similarly, the vector `num` "scatters" the result of the matrix–vector multiply to appropriate locations in `u`. A tolerance `cg_tol` enables the iterations to be stopped when successive solutions are "close enough", but since `pcg` is a loop which might carry on "forever", an iteration ceiling, `cg_limit`, is specified also. The nodal net flow rates are accumulated from element-by-element matrix–vector products in the loop called `elements_3`.

The problem given in Figure 7.20 is solved once more using Program 7.5 with the data given in Figure 7.22. The only additional data are `cg_tol` and `cg_limit`, the pcg convergence tolerance, and iteration ceiling respectively. The output in Figure 7.23 is essentially identical to that obtained previously using an assembly strategy in Figure 7.21, apart from the additional comment indicating that the pcg algorithm took 18 iterations to converge.

```
element        nod  nels  nn   nip  ndim  cg_tol   cg_limit  np_types
'hexahedron'    8    64   125   8     3   1.0e-7     200        2

prop(kx,ky,kz)
2.0  2.0  2.0
1.0  1.0  5.0

etype
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

g_coord
0.0  0.0  0.0     1.0  0.0  0.0     2.0  0.0  0.0     3.0  0.0  0.0
.
g_coord data for nodes 5-120 omitted here
.
0.0  4.0 -4.0     1.0  4.0 -4.0     2.0  4.0 -4.0     3.0  4.0 -4.0
4.0  4.0 -4.0

g_num
  6   1   2   7  31  26  27  32      7   2   3   8  32  27  28  33
.
g_num data for elements 3-62 omitted here
.
 98  93  94  99 123 118 119 124     99  94  95 100 124 119 120 125

loaded_nodes,(k,loads(k),i=1,loaded_nodes)
1
21  100.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
61
  1 0.0    2 0.0    3 0.0    4 0.0    5 0.0   10 0.0   15 0.0   20 0.0
.
fixed freedom data for 48 nodes omitted here
.
121 0.0  122 0.0  123 0.0  124 0.0  125 0.0
```

Figure 7.22 Data for Program 7.5 example

```
 There are  125 equations
 Number of cg iterations to convergence was   18

  Node Total Head    Flow rate
    1  0.0000E+00   -0.1543E+01
    2  0.0000E+00   -0.2732E+01
    3  0.0000E+00   -0.2028E+01
    4  0.0000E+00   -0.1032E+01
    5  0.0000E+00   -0.3514E+00
    6  0.3319E+01    0.1305E-05
    7  0.3183E+01    0.7655E-06
    8  0.2152E+01   -0.1016E-05
    9  0.1106E+01   -0.1038E-05
   10  0.0000E+00   -0.1026E+01
   11  0.9727E+01   -0.8685E-06
   12  0.6905E+01    0.1006E-06
   13  0.5206E+01    0.5596E-06
   14  0.2141E+01   -0.1743E-05
   15  0.0000E+00   -0.2004E+01
   16  0.1121E+02    0.2077E-05
   17  0.2098E+02   -0.1051E-05
   18  0.6877E+01    0.7030E-06
   19  0.3154E+01   -0.1291E-05
   20  0.0000E+00   -0.2684E+01
   21  0.1689E+03    0.1000E+03
.
.
.
  120  0.0000E+00   -0.2361E+00
  121  0.0000E+00   -0.5872E+00
  122  0.0000E+00   -0.8971E+00
  123  0.0000E+00   -0.7082E+00
  124  0.0000E+00   -0.3505E+00
  125  0.0000E+00   -0.1255E+00

       Inflow        Outflow
       0.1000E+03   -0.1000E+03
```

Figure 7.23 Results from Program 7.5 example

Glossary of variable names used in Chapter 7

Scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling
`fixed_down`	number of nodes on downstream side
`fixed_freedoms`	number of fixed total heads
`fixed_seep`	number of nodes on seepage surface
`fixed_up`	number of nodes on upstream side
`i`	simple counter
`iel`	simple counter
`iters`	counts free-surface iterations
`iwp`	`SELECTED_REAL_KIND(15)`

`k`	simple counter
`limit`	iteration ceiling
`loaded_nodes`	number of fixed source/sink nodes
`nci`	number of contour intervals required
`ndim`	number of dimensions
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nod`	number of nodes per element
`nn`	number of nodes in the mesh
`nprops`	number of material properties
`np_types`	number of different property types
`nxe`	number of columns of elements
`nye`	number of rows of elements

Scalar reals:

`alpha`	α from equations (3.22)
`beta`	β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`det`	determinant of the Jacobian matrix
`hdown`	fixed total head on downstream side
`hup`	fixed total head on upstream side
`d180`	set to 180.0
`initial_height`	initial height of free surface to start process
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`tol`	convergence tolerance
`up`	holds dot product $\{\mathbf{R}\}_k^T\ \{\mathbf{R}\}_k$ from equations (3.22)
`zero`	set to 0.0

Scalar characters:

`dir`	direction of element and node numbering
`element`	element type
`type_2d`	type of 2D analysis

Scalar logicals:

`cg_converged`	set to `.TRUE.` if pcg process has converged
`converged`	set to `.TRUE.` if mesh has converged

Dynamic integer arrays:

`etype`	element property types
`g_num`	global element node numbers matrix
`kdiag`	diagonal term locations
`node`	nodes with fixed total heads
`node_down`	nodes fixed on downstream side
`node_seep`	nodes fixed on downstream seepage surface

`node_up`	nodes fixed on upstream side
`num`	element node numbers

Dynamic real arrays:

`angs`	angles made by sloping mesh lines to horizontal
`bottom_width`	x-coordinates of nodes at base of mesh
`coord`	element nodal coordinates
`d`	preconditioned rhs vector
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`diag_precon`	diagonal preconditioner vector
`disps`	net nodal inflow/outflow
`ell`	element lengths
`fun`	shape functions
`gc`	integrating point coordinates
`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`kay`	permeability matrix
`kc`	element conductivity matrix
`kv`	global conductivity matrix
`kvh`	copy of `kv`
`loads`	global total head vector
`oldpot`	nodal total head values from previous iteration
`p`	"descent" vector used in equations (3.22)
`points`	integrating point local coordinates
`prop`	element properties
`store`	stores augmented diagonal terms
`storkc`	holds element conductivity matrices
`surf`	holds current total head values of free surface
`top_width`	x-coordinates of initial nodes at top of mesh
`u`	vector used in equations (3.22)
`value`	fixed values of total heads
`weights`	weighting coefficients
`x`	"old" solution vector
`xnew`	"new" solution vector
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout

7.2 Exercises

1. Steady seepage is taking place along a 1D pipe containing three porous materials with different permeabilities as indicated in Figure 7.24. The total head difference between the ends of the pipe is 100 units. Use three 1D 'rod' elements to discretise the steady flow problem and hence compute the total head values at the two intermediate

locations along the pipe and the steady flow rate through the pipe. (Ans: 27.27, 81.82, 272.2)

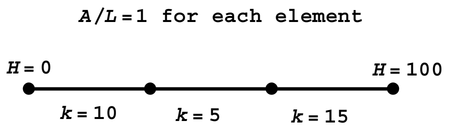


Figure 7.24

2. The square region in Figure 7.25 has anisotropic conductivity properties and boundary temperatures fixed at the values indicated. Estimate the steady state temperature at point A. (Ans: 68.3)

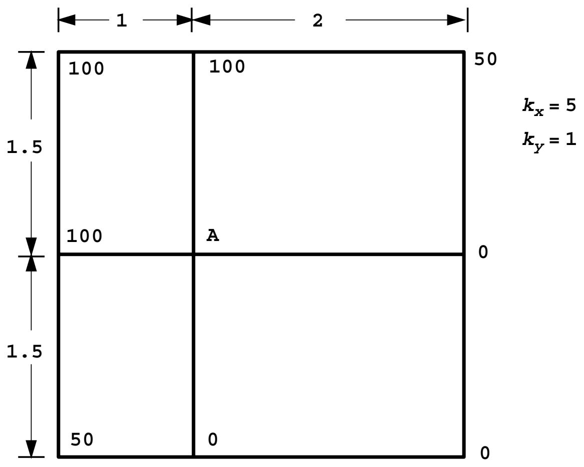


Figure 7.25

3. Steady seepage including a source (positive) is taking place along a 1D pipe containing three porous materials with different permeabilities as indicated in Figure 7.26. Use three 1D 'rod' elements to discretise the steady flow problem and hence compute the potential values at the two intermediate locations along the pipe and the net inflow/outflow through the system. (Ans: 45.45, 86.36, 454.5)

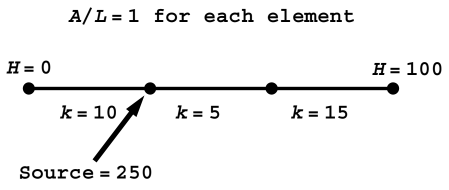


Figure 7.26

4. Compute the total head and flow rates at all the nodes in the steady flow problem shown in Figure 7.27.

(Ans: $\begin{Bmatrix} H_2 \\ H_4 \\ H_5 \end{Bmatrix} = \begin{Bmatrix} 76.67 \\ 34.05 \\ 24.76 \end{Bmatrix} \quad \begin{Bmatrix} Q_1 \\ Q_3 \\ Q_6 \end{Bmatrix} = \begin{Bmatrix} 58.3 \\ 3.6 \\ -61.9 \end{Bmatrix}$)

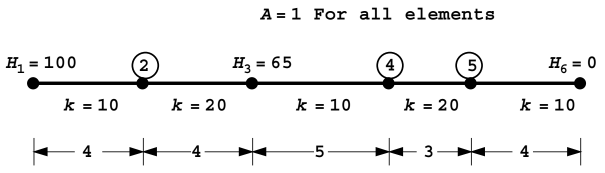


Figure 7.27

5. Steady seepage is taking place through the square block of anisotropic porous material with the external source and boundary total head values shown in Figure 7.28. Measurements indicate that $H_1 = 42.967$ and $H_2 = -1.535$. Compute the anisotropic conductivity properties k_x and k_y. (Ans: $k_x = 1$, $k_y = 2.5$)

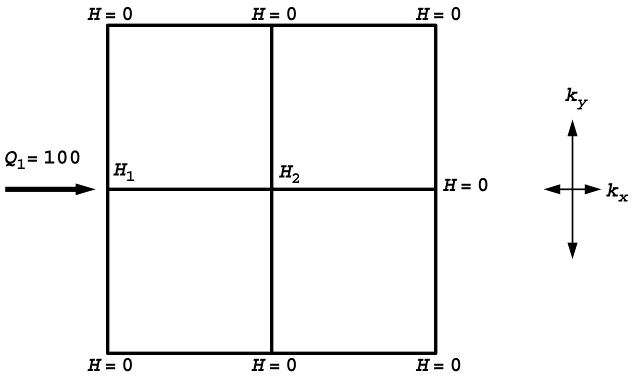


Figure 7.28

6. 1D steady seepage is taking place down a pipe as shown in Figure 7.29. Compute all the head and net flow rates at the nodes. (Ans: $H_2 = 43.69$, $H_3 = 67.76$, $Q_1 = -168.45$, $Q_4 = 193.44$)

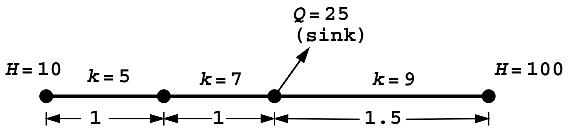


Figure 7.29

7. Steady heat flow is taking place over the square region shown in Figure 7.30 with the given boundary conditions. All elements have the same conductivity matrix given by:

$$[\mathbf{k}_c] = \begin{bmatrix} 4 & -2.5 & -2 & 0.5 \\ -2.5 & 4 & 0.5 & -2 \\ -2 & 0.5 & 4 & -2.5 \\ 0.5 & -2 & -2.5 & 4 \end{bmatrix}$$

Solve for the central temperature T. (Ans: $T = 62.5$)

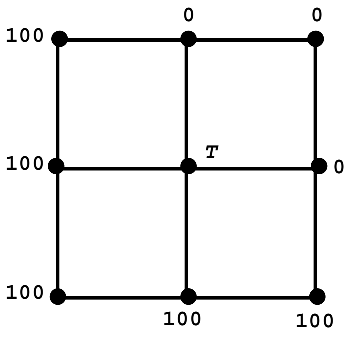


Figure 7.30

8. Use Program 7.2 to estimate the flow rate under the impermeable dam shown in Figure 7.31. (Ans: $Q \approx 25$ m^3/day/m)

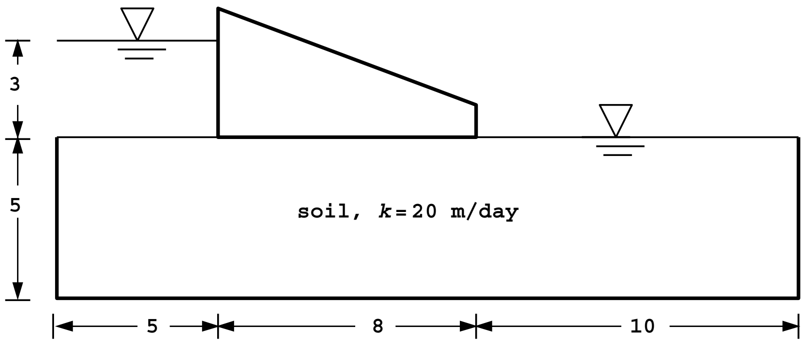


Figure 7.31

9. Use Program 7.4 to estimate the flow rate and exit hydraulic gradient due to seepage beneath the single sheet pile wall shown in Figure 7.32. (Ans: Using 50 square elements of side length 1 unit, $Q \approx 1.3 \times 10^{-5}$ m/s^3/m, $i_e \approx 0.42$)

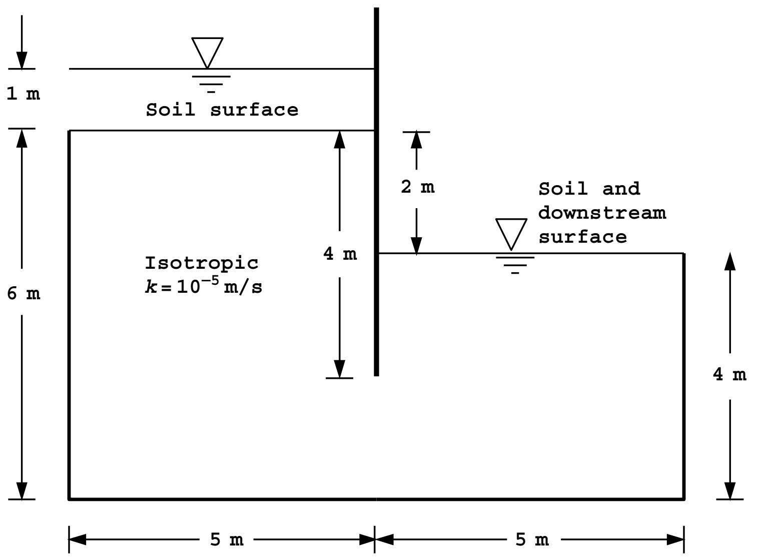


Figure 7.32

10. Use Program 7.3 to estimate the flow rate due to free-surface flow through the symmetric homogeneous embankment shown in Figure 7.33. (Ans: $Q \approx 12 \times 10^{-4}$ m^3/min/m)

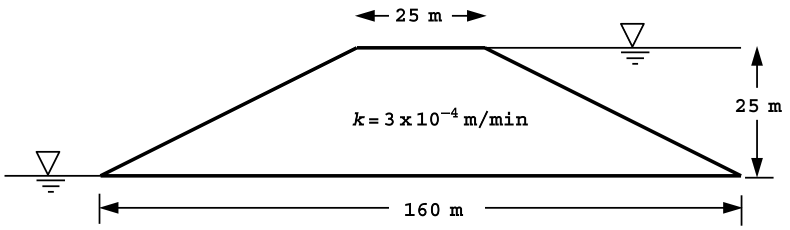


Figure 7.33

11. Derive the element conductivity matrix for a square 4-node element suitable for solving Laplace's equation for an isotropic material of permeability k.

(Ans: $[\mathbf{k}_c] = \frac{k}{6}\begin{bmatrix} 4 & -1 & -2 & -1 \\ & 4 & -1 & -2 \\ & & 4 & -1 \\ \text{symmetric} & & & 4 \end{bmatrix}$)

12. Using the matrix from the previous question, assemble the global conductivity matrix for the heat conduction problem shown in Figure 7.34 and hence solve for the steady state internal temperatures. (Ans: $T_A = 37.78,\ T_B = 10.00,\ T_C = T_D = 21.11$)

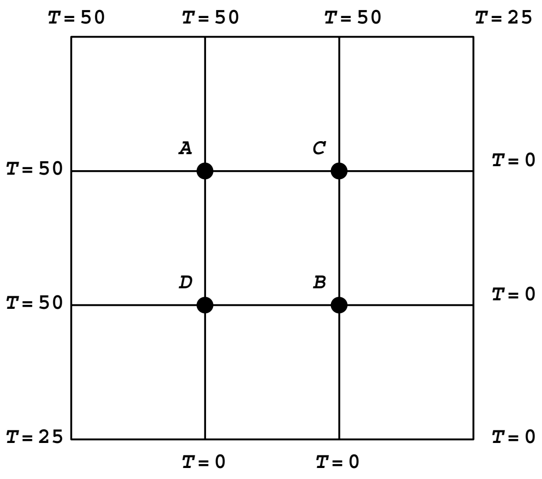


Figure 7.34

13. Derive the conductivity matrix of a 3-noded, right-angled isosceles triangular element suitable for discretisation of Laplace's equation. Use your element to estimate the steady state value of the potential at the central node of the mesh with the boundary conditions given in Figure 7.35. (Ans: 75.0)

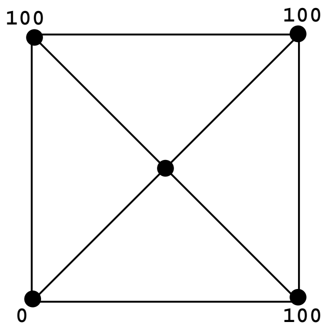


Figure 7.35

14. A square 4-node plane element of unit side length and permeability is to be used in the solution of Laplace's equation over a two-dimensional isotropic medium. If the terms of the element conductivity matrix can be expressed in the form:

$$k_{ij} = \int_0^1 \int_0^1 f_{ij}(x, y)\ \mathrm{d}x\,\mathrm{d}y, \quad i, j = 1, 2, 3, 4$$

find the function f_{14} and evaluate k_{14} explicitly.
(Ans: $f_{14} = -(1-y)^2 + x(1-x)$, $k_{14} = -\frac{1}{6}$)

References

Griffiths DV 1984 Rationalised charts for the method of fragments applied to confined seepage. *Géotechnique* **34**(2), 229–238.

Griffiths DV 1991 Generalised numerical integration of moments. *Int J Numer Methods Eng* **32**(1), 129–147.

Verruijt A 1970 *Theory of Groundwater Flow*. Macmillan, London.

8

Transient Problems: First Order (Uncoupled)

8.1 Introduction

In the previous chapter, programs for the solution of steady state potential flow problems were described. Typically, Laplace's equation (2.122) was discretised in space into an equilibrium equation (2.123) involving the solution of a set of simultaneous equations. For well-posed problems there are usually no associated numerical difficulties.

When a flow process is transient, or time dependent, the simplest extension of equation (2.122), or reduction of the Navier–Stokes equations, is provided by equations like (2.130). There is still a single dependent variable (for example potential), and so the analysis is "uncoupled". After discretisation in space, a typical element equation is given by equation (2.131). Problems such as these lead to a set of first order ordinary differential equations in time, the solution of which is no longer a simple numerical task for large numbers of elements.

Some of the many solution techniques available were described in Chapter 3. Possibly the simplest, and most robust, are the "implicit" methods described by equation (3.94) and by the structure chart in Figure 3.19. These θ-methods form the basis of the first three programs in this chapter. Program 8.1 analyses the 1D consolidation equation, and Programs 8.2 and 8.3 extend this approach to two dimensions. In the case of Program 8.3, a "mesh-free" preconditioned conjugate gradient approach is used. Program 8.4 demonstrates an "explicit" ($\theta = 0$) solution strategy to a 2D transient problem, and Program 8.5 uses an "element-by-element" operator splitting method. Program 8.6 returns to the "implicit" θ methods allowing analysis of 2D or 3D transient problems over a general finite element mesh. The final two Programs 8.7 and 8.8 solve the diffusion–convention equation (2.132) in 2D using, respectively, "transformed" and "untransformed" analyses.

Program 8.1 One-dimensional consolidation analysis using 2-node line elements. Implicit time integration using the "theta" method.

```
PROGRAM p81
!-------------------------------------------------------------------------
! Program 8.1 One dimensional consolidation analysis using 2-node line
!             elements. Implicit time integration using the "theta" method.
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,j,nels,neq,nod=2,npri,nprops=1,np_types,   &
   nres,nstep,ntime
 REAL(iwp)::at,a0,dtim,penalty=1.0e20_iwp,pt5=0.5_iwp,theta,time,         &
   zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),node(:),num(:),kdiag(:)
 REAL(iwp),ALLOCATABLE::bp(:),ell(:),kc(:,:),kv(:),loads(:),newlo(:),     &
   mm(:,:),press(:),prop(:,:),storbp(:),value(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,np_types; neq=nels+1
 ALLOCATE(num(nod),etype(nels),kc(nod,nod),mm(nod,nod),press(0:neq),      &
   prop(nprops,np_types),ell(nels),kdiag(neq),loads(0:neq),newlo(0:neq))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
 READ(10,*)dtim,nstep,theta,npri,nres,ntime; kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   num=(/iel,iel+1/); CALL fkdiag(kdiag,num)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq))); bp=zero; kv=zero
 WRITE(11,'(2(a,i5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----------------------global conductivity and "mass" matrix assembly----
 elements_2: DO iel=1,nels
   num=(/iel,iel+1/)
   CALL rod_km(kc,prop(1,etype(iel)),ell(iel)); CALL rod_mm(mm,ell(iel))
   CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
 END DO elements_2; kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta
!-----------------------specify initial and boundary values---------------
 loads(0)=zero; READ(10,*)loads(1:); a0=zero
 DO iel=1,nels; a0=a0+pt5*ell(iel)*(loads(iel)+loads(iel+1)); END DO
 READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)then
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),                   &
            storbp(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
 END IF
!-----------------------factorise equations-------------------------------
 CALL sparin(bp,kdiag)
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/a,i3,a)')"    Time      Deg of Con  Pressure (node",nres,")"
 WRITE(11,'(3e12.4)')0.0,0.0,loads(nres)
 timesteps: DO j=1,nstep
   time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag)
   IF(fixed_freedoms/=0)newlo(node)=storbp*value
   CALL spabac(bp,newlo,kdiag); loads=newlo; at=zero
   DO iel=1,nels; at=at+pt5*ell(iel)*(loads(iel)+loads(iel+1)); END DO
   IF(j==ntime)press(1:)=loads(1:)
   IF(j/npri*npri==j)WRITE(11,'(3e12.4)')time,(a0-at)/a0,loads(nres)
 END DO timesteps
 WRITE(11,'(/a,e10.4,a)')"    Depth     Pressure (time=",ntime*dtim,")"
 WRITE(11,'(3e12.4)')0.0,press(1)
 WRITE(11,'(2e12.4)')(SUM(ell(1:i)),press(i+1),i=1,nels)
STOP
END PROGRAM p81
```

Scalar integers:

`fixed_freedoms`	number of fixed nodes
`i`	simple counter
`iel`	simple counter
`iwp`	`SELECTED_REAL_KIND(15)`
`j`	simple counter
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nod`	number of nodes per element
`npri`	output printed every `npri` time steps
`nprops`	number of material properties
`np_types`	number of different property types
`nres`	node number at which time history is to be printed
`nstep`	number of time steps required
`ntime`	time step number at which spatial distribution is to be printed

Scalar reals:

`at`	holds area beneath isochrone by Trapezoid Rule at time t
`a0`	holds area beneath isochrone by Trapezoid Rule at time $t = 0$
`dtim`	calculation time step
`pt_5`	set to 0.5
`penalty`	set to 1×10^{20}
`theta`	time integration weighting parameter
`time`	holds elapsed time t
`zero`	set to 0.0

Dynamic integer arrays:

`etype`	element property types
`kdiag`	diagonal term locations
`node`	nodes with fixed values
`num`	element node numbers

Dynamic real arrays:

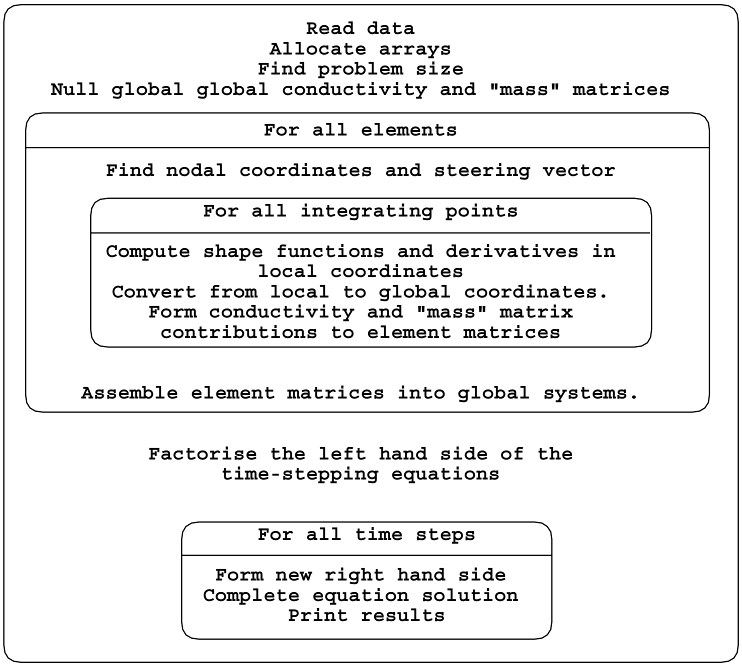
`bp`	global “mass” matrix
`ell`	element lengths
`kc`	element conductivity matrix
`kv`	global conductivity matrix
`loads`	excess pore pressure values

`newlo`	new excess pore pressure values
`mm`	element "mass" matrix
`press`	excess pore pressure values after `ntime` time steps
`prop`	element properties
`storbp`	copy of `bp`
`value`	fixed boundary values of excess pore pressure

In the absence of sources or sinks, equation (3.94) reduces to

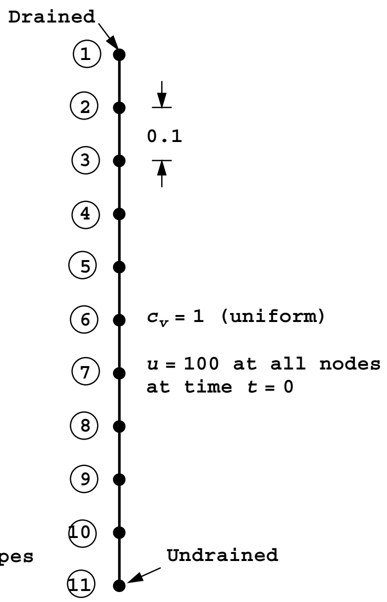
$$([\mathbf{M}_m] + \theta \Delta t\, [\mathbf{K}_c])\,\{\mathbf{\Phi}\}_1 = ([\mathbf{M}_m] - (1-\theta)\Delta t\, [\mathbf{K}_c])\,\{\mathbf{\Phi}\}_0 \tag{8.1}$$

Where the element conductivity $[\mathbf{k}_c]$ and "mass" $[\mathbf{m}_m]$ matrices (`kc` and `mm` respectively in programming terminology) have been assembled into their global counterparts $[\mathbf{K}_c]$ and $[\mathbf{M}_m]$ (`kv` and `bp`). After some manipulations, the left hand side matrix $([\mathbf{M}_m] + \theta \Delta t\, [\mathbf{K}_c])$ is formed (called `bp`) and the fixed boundary conditions are implemented using the "stiff spring" or "penalty" strategy. The matrix is then factorised by subroutine `sparin`, the initial conditions are read into $\{\mathbf{\Phi}\}_0$ (`loads`), and the right-hand side matrix-by-vector product $([\mathbf{M}_m] - (1-\theta)\Delta t\, [\mathbf{K}_c])\,\{\mathbf{\Phi}\}_0$ (`newlo`) computed. Within each time step, all that is then required to advance the solution, is a matrix-by-vector multiplication on the right hand side of equation (8.1), followed by a forward and backward substitution. The final section of the program consists of the time-stepping loop completed `nstep` times. The matrix-by-vector multiplication is carried out by `linmul_sky` and forward and backward substitution by `spabac`. The process is described in detail by the structure chart in



```
Read data
Allocate arrays
Find problem size
Null global global conductivity and "mass" matrices
For all elements
    Find nodal coordinates and steering vector
    For all integrating points
        Compute shape functions and derivatives in
        local coordinates
        Convert from local to global coordinates.
        Form conductivity and "mass" matrix
        contributions to element matrices
    Assemble element matrices into global systems.
Factorise the left hand side of the
time-stepping equations
For all time steps
    Form new right hand side
    Complete equation solution
    Print results
```

Figure 8.1 Structure chart for implicit analysis of transient problems with assembly



```
nels  np_types
10      1

prop(cv)
1.0

etype(not needed)

ell
0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1

dtim  nstep  theta  npri  nres  ntime
0.001 2000    0.5   100    11   1000

loads(i),i=1,neq
100.0  100.0  100.0  100.0  100.0  100.0
100.0  100.0  100.0  100.0  100.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
1
1 0.0
```

Figure 8.2 Mesh and data for Program 8.1 example

Figure 8.1. Note, however, that the matrix-by-vector multiplication on the right hand side could be done using element-by-element summation, avoiding storage of one large matrix.

Figure 8.2 shows a string of 10 elements attached end to end, representing a 1D layer of saturated soil with a total depth of 1.0. The layer is subjected to a uniform initial excess pore pressure distribution of 100.0 and is drained at the top only. The material property required in this analysis is the coefficient of consolidation c_v (analogous to *EA* in Program 4.1 and *kA* in Program 7.1). The objective of the analysis is to compute the excess pore pressure distribution and the average degree of consolidation as a function of time.

The data involve reading the number of elements `nels=10`, and the number of property types `np_types=1`. In this case the layer is uniform, so with `np_types=1` the `etype` data is not needed. The 10 element lengths `ell` are read, and in this example are all the same length and equal to 0.1.

The next line of data refers to the time-stepping and the output parameters. The three time-stepping parameters are the calculation time step `dtim`, read as 0.001, the number of calculation time steps required `nstep`, read as 2,000 and the time integration parameters `theta`, read next as 0.5. The choice of $\theta = 0.5$ is often referred to as the "Crank–Nicolson" method of time integration. The three output parameters are the output frequency parameter `npri`, read as 100, the node at which a time history is required `nres`, read as 11 (corresponding to the bottom node at the impermeable boundary) and the time step at which a spatial distribution of excess pore pressure is required `ntime`, read as 1,000 and corresponding to $t = 1.0$. The user could of course modify the program to generate alternative output if required.

The next data read into `loads`, gives the initial excess pore pressure at the 11 nodes at $t = 0.0$, which in this example is uniform and equal to 100.0. The final data gives the drainage boundary conditions. In this example, one node (node 1) is a drainage boundary, so its excess pore pressure is fixed equal to zero. Users are invited to try other initial conditions and boundary conditions (e.g. linear variation of excess pore pressure, double drainage, etc.).

The results given in Figure 8.3 show the time history of the excess pore pressure at node 11 and the average degree of consolidation (U) every 100 time steps up to $t = 2.0$. The lower part gives the excess pore pressure with depth corresponding to $t = 1.0$. With the maximum drainage path and the coefficient of consolidation both equal to unity in this example, the dimensionless time factor equals real time, thus $T = t$.

```
There are   11 equations and the skyline storage is   21

   Time        Deg of Con  Pressure (node 11)
 0.0000E+00  0.0000E+00  0.1000E+03
 0.1000E+00  0.3567E+00  0.9525E+02
 0.2000E+00  0.5041E+00  0.7743E+02
 0.3000E+00  0.6134E+00  0.6079E+02
 0.4000E+00  0.6981E+00  0.4751E+02
 0.5000E+00  0.7643E+00  0.3711E+02
 0.6000E+00  0.8159E+00  0.2898E+02
 0.7000E+00  0.8562E+00  0.2263E+02
 0.8000E+00  0.8877E+00  0.1767E+02
 0.9000E+00  0.9123E+00  0.1380E+02
 0.1000E+01  0.9315E+00  0.1078E+02
 0.1100E+01  0.9465E+00  0.8417E+01
 0.1200E+01  0.9582E+00  0.6574E+01
 0.1300E+01  0.9674E+00  0.5134E+01
 0.1400E+01  0.9745E+00  0.4009E+01
 0.1500E+01  0.9801E+00  0.3131E+01
 0.1600E+01  0.9845E+00  0.2445E+01
 0.1700E+01  0.9879E+00  0.1909E+01
 0.1800E+01  0.9905E+00  0.1491E+01
 0.1900E+01  0.9926E+00  0.1165E+01
 0.2000E+01  0.9942E+00  0.9094E+00

   Depth       Pressure (time=0.1000E+01)
 0.0000E+00 -0.1967E-21
 0.1000E+00  0.1686E+01
 0.2000E+00  0.3331E+01
 0.3000E+00  0.4893E+01
 0.4000E+00  0.6335E+01
 0.5000E+00  0.7621E+01
 0.6000E+00  0.8720E+01
 0.7000E+00  0.9604E+01
 0.8000E+00  0.1025E+02
 0.9000E+00  0.1065E+02
 0.1000E+01  0.1078E+02
```

Figure 8.3 Results from Program 8.1 example

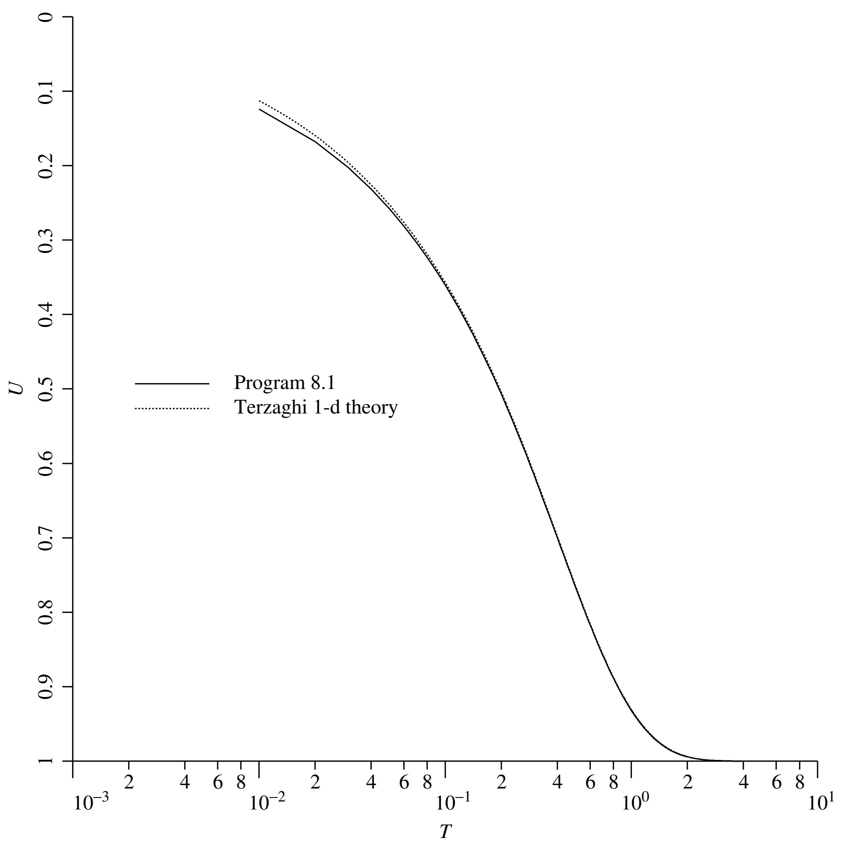


Figure 8.4 Comparison of Program 8.1 results with Terzaghi's consolidation theory

Figure 8.4 gives a plot of the computed U vs. T, showing excellent agreement with the series solution from Terzaghi's 1D consolidation theory.

Program 8.2 Plane or axisymmetric consolidation analysis using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$- or $y(z)$-direction. Implicit time integration using the "theta" method.

```
PROGRAM p82
!-------------------------------------------------------------------------
! Program 8.2 Plane or axisymmetric consolidation analysis using 4-node
!             rectangular quadrilaterals. Mesh numbered in x(r)- or y(z)-
!             direction. Implicit time integration using the "theta"
!             method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,j,nci,ndim=2,nels,neq,nip=4,nn,nod=4,npri, &
   np_types,nres,nstep,ntime,nxe,nye
 REAL(iwp)::det,dtim,one=1.0_iwp,penalty=1.0e20_iwp,theta,time,          &
   zero=0.0_iwp; CHARACTER(len=15)::dir,element='quadrilateral',type_2d
```

```
!-----------------------dynamic arrays----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),node(:),num(:),kdiag(:)
 REAL(iwp),ALLOCATABLE::bp(:),coord(:,:),der(:,:),deriv(:,:),fun(:),      &
   gc(:),g_coord(:,:),jac(:,:),kay(:,:),kc(:,:),kv(:),loads(:),newlo(:),  &
   ntn(:,:),mm(:,:),points(:,:),prop(:,:),storbp(:),value(:),weights(:),  &
   x_coords(:),y_coords(:)
!-------------------------input and initialisation-----------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)type_2d,dir,nxe,nye,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
 ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim),   &
   fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),&
   mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod),         &
   etype(nels),kdiag(neq),loads(0:neq),newlo(0:neq),x_coords(nxe+1),      &
   y_coords(nye+1),prop(ndim,np_types),gc(ndim))
 READ(10,*)prop; etype=1; if(np_types>1)read(10,*)etype
 READ(10,*)x_coords,y_coords
 READ(10,*)dtim,nstep,theta,npri,nres,ntime; kdiag=0
! ----------loop the elements to set up global geometry and kdiag --------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
   CALL fkdiag(kdiag,num)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq)))
 WRITE(11,'(2(a,i5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); bp=zero; kv=zero; gc=one
!-----------------------global conductivity and "mass" matrix assembly----
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
   gauss_pts: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); if(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
     kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
     CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
   END DO gauss_pts
   CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
 END DO elements_2
 kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta
!-----------------------specify initial and boundary values---------------
 READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)then
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),                   &
            storbp(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
 END IF
!-----------------------factorise equations-------------------------------
 CALL sparin(bp,kdiag)
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/a,i3,a)')"    Time      Pressure (node",nres,")"
 WRITE(11,'(2e12.4)')0.0,loads(nres)
 timesteps: DO j=1,nstep
   time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag)
   IF(fixed_freedoms/=0)newlo(node)=storbp*value
   CALL spabac(bp,newlo,kdiag); loads=newlo
   IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
     CALL contour(loads,g_coord,g_num,nci,13); END IF
   IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
 END DO timesteps
STOP
END PROGRAM p82
```

New scalar integers:

`nci`	number of contour intervals
`ndim`	number of dimensions
`nip`	number of integrating points
`nn`	number of nodes in the mesh
`nxe`	number of elements in $x(r)$-direction
`nye`	number of elements in $y(z)$-direction

New scalar reals:

`det`	determinant of the Jacobian matrix
`one`	set to 1.0

Scalar characters:

`dir`	element and node numbering direction
`element`	element type
`type_2d`	type of 2D analysis (`'plane'` or `'axisymmetric'`)

New dynamic integer arrays:

`g_num`	global element node numbers matrix

New dynamic real arrays:

`coord`	element nodal coordinates
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`fun`	shape functions
`gc`	integrating point coordinates
`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`kay`	permeability matrix
`points`	integrating point local coordinates
`weights`	weighting coefficients
`x_coords`	$x(r)$-coordinates of mesh layout
`y_coords`	$y(z)$-coordinates of mesh layout

This program is for the analysis of 2D (`ndim=2`) first-order transient problems under plane or axisymmetric conditions, and is closely based on Program 7.2 in Chapter 7. In order to simplify the data however, the examples presented here use 4-node rectangular elements only (`element='quadrilateral'` and `nod=4`). The program includes graphics subroutines `mesh` and `contour` which generate PostScript files containing, respectively, images of the finite element mesh (held in `fe95.msh`), and a contour map of the excess pore pressure (held in the fe95.con) corresponding to the solution at the requested time step ntime. The contouring only works currently for meshes made up of 4-node quadrilateral elements.

The first example chosen is shown in Figure 8.5 and could represent dissipation of excess porewater pressure from a square block of soil with drainage permitted at the outside boundaries. Due to 4-fold symmetry only one-quarter of the square is modelled with excess pore pressure at the outer drained boundaries fixed to zero, and the two inner boundaries defaulting to "no-flow" boundary conditions.

The first line of data reads type_2d and dir and indicates that a plane analysis is to be performed with element and node numbering in the x-direction. The second line shows that the rectangular mesh consists of 5 columns (nxe) and 5 rows (nye) of elements, and there is only one property type (np_types) in this homogeneous example. The third line reads the 2D coefficients of consolidation c_x and c_y into the property array prop, and since there is only one property type in this problem, the etype data is not required. The fourth and fifth lines give, respectively, the x-(x_coords) and y-coordinates (y_coords) of the lines that make up the mesh. The sixth line of data reads the time-stepping and output parameters with the same meaning as in the data for Program 8.1. The next data read into loads, is the initial excess pore pressure at all the nodes in the mesh. In this example, the square block is subjected to an initial uniform excess pore pressure of 100.0, with drainage immediately effective at the top and right boundaries. The next line of data indicates that there are 11 (fixed_freedoms) nodes to have fixed values. The next two lines of data

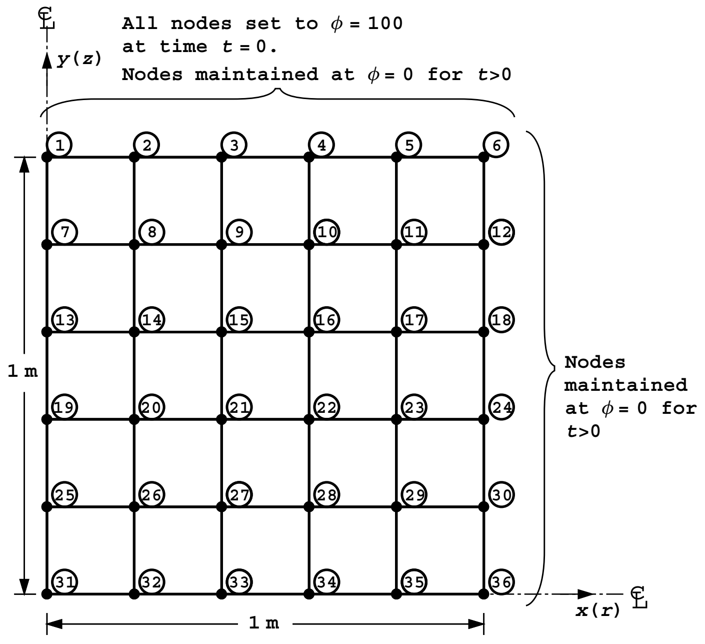


Figure 8.5 Mesh and data for the first Program 8.2 and 8.5 example (*Continued on page 367*)

```
type_2d  dir
'plane'  'x'

nxe  nye  np_types
5     5     1

prop(cx,cy)
1.0  1.0

etype(not needed)

x_coords, y_coords
 0.0  0.2  0.4  0.6  0.8  1.0
 0.0 -0.2 -0.4 -0.6 -0.8 -1.0

dtim  nstep  theta  npri  nres  ntime
0.01   150    0.5    10    31    100

loads(i),i=1,neq
  0.0    0.0    0.0    0.0    0.0    0.0
100.0 100.0 100.0 100.0 100.0    0.0
100.0 100.0 100.0 100.0 100.0    0.0
100.0 100.0 100.0 100.0 100.0    0.0
100.0 100.0 100.0 100.0 100.0    0.0
100.0 100.0 100.0 100.0 100.0    0.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
11
 1  0.0    2  0.0    3  0.0    4  0.0    5  0.0    6  0.0
12  0.0   18  0.0   24  0.0   30  0.0   36  0.0

nci
10
```

Figure 8.5 (*Continued from page 366*)

```
There are   36 equations and the skyline storage is  246

     Time       Pressure (node 31)
  0.0000E+00  0.1000E+03
  0.1000E+00  0.9009E+02
  0.2000E+00  0.5845E+02
  0.3000E+00  0.3580E+02
  0.4000E+00  0.2178E+02
  0.5000E+00  0.1324E+02
  0.6000E+00  0.8051E+01
  0.7000E+00  0.4895E+01
  0.8000E+00  0.2976E+01
  0.9000E+00  0.1809E+01
  0.1000E+01  0.1100E+01
  0.1100E+01  0.6687E+00
  0.1200E+01  0.4065E+00
  0.1300E+01  0.2472E+00
  0.1400E+01  0.1503E+00
  0.1500E+01  0.9135E-01
```

Figure 8.6 Results from first Program 8.2 example

indicate the node numbers (`node`) and the values (`value`) to which they are to be fixed (zero in the case of drainage boundaries). The final line of data reads `nci`, indicating that the contour map of excess pore pressure, corresponding to the situation after `ntime` time steps and written to file `fe95.con`, will include 10 contour intervals.

The output shown in Figure 8.6 gives the excess pore pressure at node `nres=31`, which is the centre of the mesh, every `npri=10` time steps (every 0.1 s). The normalised result at

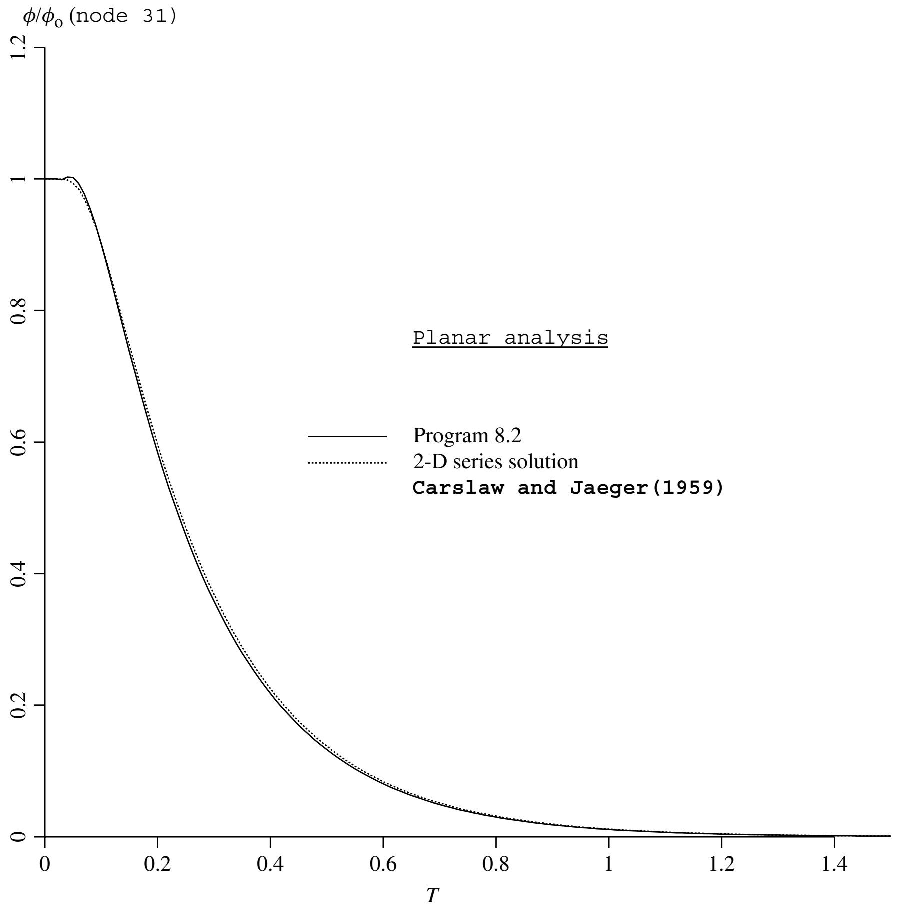


Figure 8.7 Comparison of Program 8.2 result for a planar region with 2D series solution

node 31 (after division by the initial value) is plotted against the time factor T in Figure 8.7, where it is compared with series solution values (Carslaw and Jaeger, 1959). The crude finite element idealisation gives excellent agreement. The contour map corresponding to the distribution of excess pore pressure after `ntime` time steps ($t = 1.0$ secs) is given in Figure 8.8.

The second example is of the same problem considered previously, but under axisymmetric conditions. In soil mechanics, the physical analogue would be a "triaxial" specimen of soil draining from all its boundaries.

The data shown in Figure 8.9 is very similar to that of the previous example, but with `type_2d` set to `'axisymmetric'` and `dir` set to `'r'`, because the node and element numbering is now in the radial direction. The number of integrating points `nip` remains equal to 4 for the rectangular 4-node elements considered in this example (see discussion of `nip` in Program 7.2). The output at node 31 shown in Figure 8.10, is plotted in Figure 8.11 and compared again with Carslaw and Jaeger's (1959) axisymmetric solution.

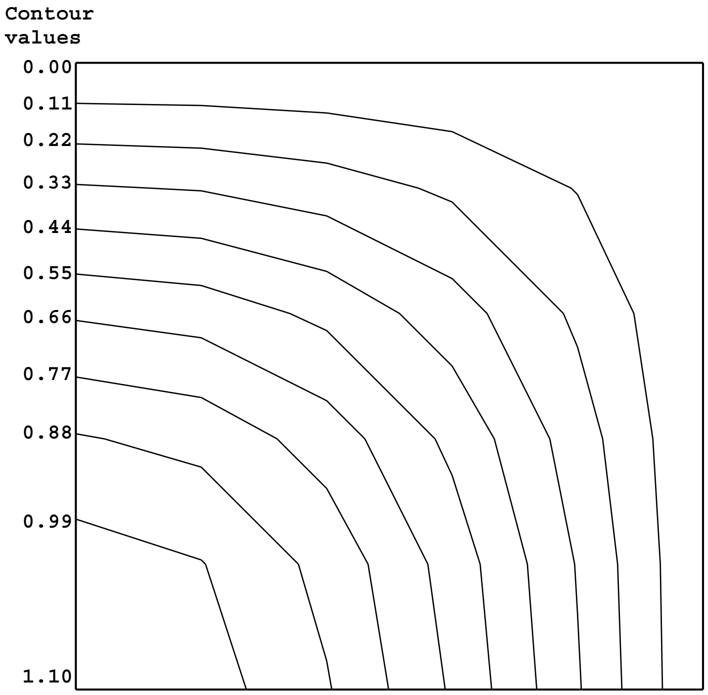


Figure 8.8 Contour map of excess pore pressure after $t = 1.0$ from first Program 8.2 example

```
type_2d  dir
'axisymmetric'  'r'

nxe  nye  np_types
5    5    1

prop(cx,cy)
1.0  1.0

etype(not needed)

x_coords, y_coords
 0.0  0.2  0.4  0.6  0.8  1.0
 0.0 -0.2 -0.4 -0.6 -0.8 -1.0

dtim  nstep  theta  npri  nres  ntime
0.01   150    0.5    10    31    100

loads(i),i=1,neq
  0.0   0.0   0.0   0.0   0.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
11
 1  0.0    2  0.0    3  0.0    4  0.0    5  0.0    6  0.0
12  0.0   18  0.0   24  0.0   30  0.0   36  0.0

nci
10
```

Figure 8.9 Data for the second Program 8.2 example

```
 There are   36 equations and the skyline storage is  246

    Time       Pressure (node 31)
  0.0000E+00  0.1000E+03
  0.1000E+00  0.8096E+02
  0.2000E+00  0.3798E+02
  0.3000E+00  0.1667E+02
  0.4000E+00  0.7280E+01
  0.5000E+00  0.3176E+01
  0.6000E+00  0.1386E+01
  0.7000E+00  0.6046E+00
  0.8000E+00  0.2638E+00
  0.9000E+00  0.1151E+00
  0.1000E+01  0.5022E-01
  0.1100E+01  0.2191E-01
  0.1200E+01  0.9560E-02
  0.1300E+01  0.4171E-02
  0.1400E+01  0.1820E-02
  0.1500E+01  0.7940E-03
```

Figure 8.10 Results from second Program 8.2 example

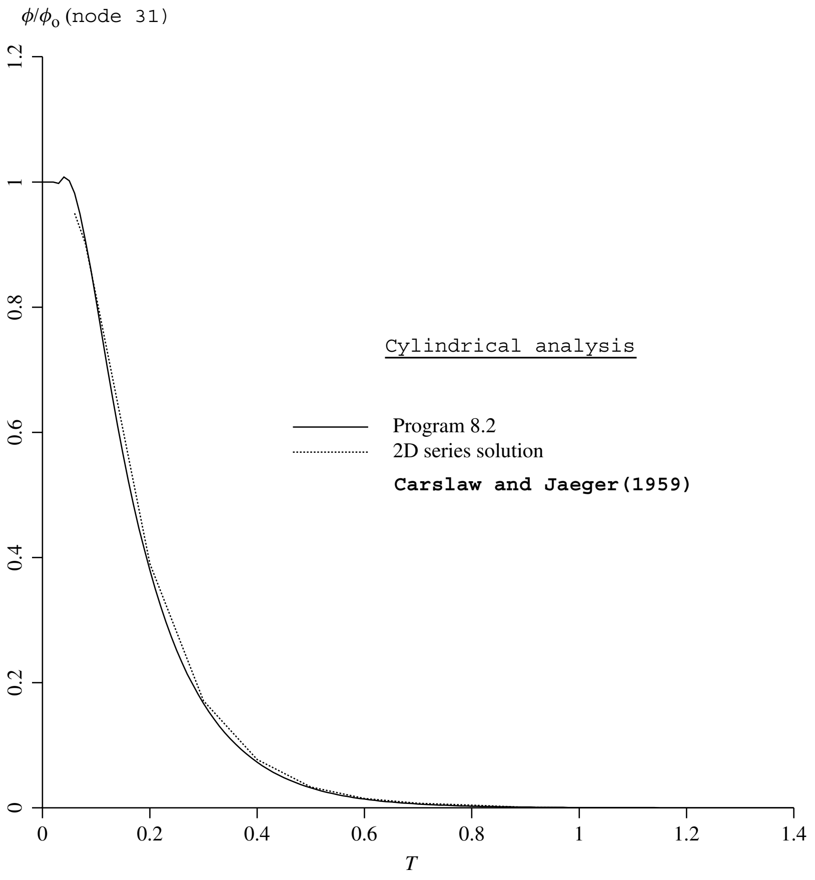


Figure 8.11 Comparison of Program 8.2 result for a cylindrical region with series solution

8.2 Mesh-free Strategies in Transient Analysis

The first two programs in this Chapter have used implicit integration in time and assembly strategies for mesh conductivity and “mass” matrices. For very large problems, the demands on computer storage become significant and, as was the case in Chapters 5 to 7, it is natural to seek “mesh-free” strategies which avoid the need to store system matrices at all.

Possibly the simplest (and oldest) mesh-free strategy is based on equation (3.98) which shows that for lumped “mass” matrix [$\mathbf{M}_m$], the solution at a new time can be found from the solution at the previous time by a simple matrix–vector multiplication, which can be done element-by-element. Such “explicit” techniques suffer from numerical stability difficulties and are described later. First we consider the natural extension to Programs 8.1 and 8.2, which were seen to consist of a linear equation solution on every time step. Clearly this solution can be accomplished iteratively using pcg or some similar technique, and this is done in Program 8.3.

Program 8.3 Plane or axisymmetric consolidation analysis using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$- or $y(z)$-direction. Implicit time integration using the “theta” method. No global stiffness matrix assembly. Diagonal preconditioner conjugate gradient solver.

```
PROGRAM p83
!-------------------------------------------------------------------------
! Program 8.3 Plane or axisymmetric consolidation analysis using 4-node
!             rectangular quadrilaterals. Mesh numbered in x(r)- or y(z)-
!             direction. Implicit time integration using the "theta"
!             method. No global matrix assembly. Diagonal
!             preconditioner conjugate gradient solver
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,j,k,nci,ndim=2,nels,neq, &
   nip=4,nn,nod=4,npri,np_types,nres,nstep,ntime,nxe,nye
 REAL(iwp)::alpha,beta,cg_tol,det,dtim,one=1.0_iwp,penalty=1.0e20_iwp,    &
   theta,time,up,zero=0.0_iwp; LOGICAL::cg_converged
 CHARACTER(LEN=15)::dir,element='quadrilateral',type_2d
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),d(:),der(:,:),deriv(:,:),              &
   diag_precon(:),fun(:),gc(:),g_coord(:,:),jac(:,:),kay(:,:),kc(:,:),    &
   loads(:),ntn(:,:),p(:),mm(:,:),points(:,:),prop(:,:),r(:),store(:),    &
   storka(:,:,:),storkb(:,:,:),u(:),value(:),weights(:),x(:),xnew(:),     &
   x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)type_2d,dir,nxe,nye,cg_tol,cg_limit,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
 WRITE(11,'(a,i5,a)')" There are",neq," equations"
 ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim),   &
   fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),&
   mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod),         &
   storka(nod,nod,nels),storkb(nod,nod,nels),etype(nels),x_coords(nxe+1), &
   y_coords(nye+1),prop(ndim,np_types),loads(0:neq),diag_precon(0:neq),   &
   u(0:neq),d(0:neq),p(0:neq),x(0:neq),r(0:neq),xnew(0:neq),gc(ndim))
 READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
 READ(10,*)x_coords,y_coords
 READ(10,*)dtim,nstep,theta,npri,nres,ntime
!---------------loop the elements to set up element data------------------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 CALL sample(element,points,weights); diag_precon=zero; gc=one
!----------element matrix integration, storage and preconditioner------
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
   gauss_pts: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); IF(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
     kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
     CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
   END DO gauss_pts
   storka(:,:,iel)=mm+kc*theta*dtim; storkb(:,:,iel)=mm-kc*(one-theta)*dtim
   DO k=1,nod
     diag_precon(num(k))=diag_precon(num(k))+storka(k,k,iel)
   END DO
 END DO elements_2
!-----------------------specify initial and boundary values---------------
 READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),                   &
            store(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   diag_precon(node)=diag_precon(node)+penalty; store=diag_precon(node)
 END IF
 diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/a,i3,a)')"    Time      Pressure (node",nres,")  cg iters"
 WRITE(11,'(2e12.4)')0.0,loads(nres)
 timesteps: DO j=1,nstep
   time=j*dtim; u=zero
   elements_3 : DO iel=1,nels
     num=g_num(:,iel); kc=storkb(:,:,iel)
     u(num)=u(num)+MATMUL(kc,loads(num))
   END DO elements_3
   u(0)=zero; r=u; IF(fixed_freedoms/=0)r(node)=store*value
   d=diag_precon*r; p=d; x=zero; cg_iters=0
!-----------------------pcg equation solution-----------------------------
   pcg: DO
      cg_iters=cg_iters+1; u=zero
      elements_4: DO iel=1,nels
        num=g_num(:,iel); kc=storka(:,:,iel)
        u(num)=u(num)+MATMUL(kc,p(num))
      END DO elements_4
      IF(fixed_freedoms/=0)u(node)=p(node)*store; up=DOT_PRODUCT(r,d)
      alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha; r=r-u*alpha
      d=diag_precon*r; beta=DOT_PRODUCT(r,d)/up; p=d+p*beta
     CALL checon(xnew,x,cg_tol,cg_converged)
      IF(cg_converged.OR.cg_iters==cg_limit)EXIT
   END DO pcg; loads=xnew
   IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
     CALL contour(loads,g_coord,g_num,nci,13); END IF
   IF(j/npri*npri==j)                                                    &
     WRITE(11,'(2e12.4,7x,i5)')time,loads(nres),cg_iters
 END DO timesteps
STOP
END PROGRAM p83
```

New scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling
`k`	simple counter

New scalar integers:

`alpha`	α from equations (3.22)
`beta`	β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\{\mathbf{R}\}_k$ from equations (3.22)

Scalar logical:

`cg_converged`	set to `.TRUE.` if pcg process has converged

New dynamic real arrays:

`d`	vector used in equation (3.22)
`diag_precon`	diagonal preconditioner vector
`p`	"descent" vector used in equations (3.22)
`r`	holds fixed rhs terms in pcg solver
`store`	stores global augmented diagonal terms
`storka`	stores lhs element matrix
`storkb`	stores rhs element matrix
`u`	vector used in equation (3.22)
`x`	"old" solution vector
`xnew`	"new" solution vector

In this program, all the element conductivity and "mass" matrices are stored in the forms $([\mathbf{m}_m] + \theta\Delta t[\mathbf{k}_c])$ and $([\mathbf{m}_m] - (1-\theta)\Delta t[\mathbf{k}_c])$ in arrays `storka` and `storkb` respectively, as required by equation (3.94). The diagonal preconditioner is formed from the diagonal terms of the first of these as it would have been assembled. The only additional inputs compared to Program 8.2 are the iteration tolerance, `cg_tol`, and the limiting number of pcg iterations, `pcg_limit`. The data are shown as Figure 8.12 with output as Figure 8.13. For an iteration tolerance of 0.0001 and the same time step as was used in Program 8.2, the pcg process converges in at most 3 iterations, and leads to the same solution as given in Figure 8.6.

```
type_2d  dir
'plane'  'x'

nxe  nye  cg_tol  cg_limit  np_types
5     5   0.0001    100       1

prop(cx,cy)
1.0  1.0

etype(not needed)

x_coords, y_coords
 0.0  0.2  0.4  0.6  0.8  1.0
 0.0 -0.2 -0.4 -0.6 -0.8 -1.0

dtim  nstep  theta  npri  nres  ntime
0.01   150    0.5    10    31    100

loads(i),i=1,neq
  0.0   0.0   0.0   0.0   0.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
11
 1  0.0    2  0.0    3  0.0    4  0.0    5  0.0    6  0.0
12  0.0   18  0.0   24  0.0   30  0.0   36  0.0

nci
10
```

Figure 8.12 Data for Program 8.3 example

```
There are   36 equations

     Time      Pressure (node 31)  cg iters
   0.0000E+00  0.1000E+03
   0.1000E+00  0.9009E+02           3
   0.2000E+00  0.5845E+02           3
   0.3000E+00  0.3580E+02           2
   0.4000E+00  0.2178E+02           2
   0.5000E+00  0.1324E+02           2
   0.6000E+00  0.8051E+01           2
   0.7000E+00  0.4895E+01           2
   0.8000E+00  0.2976E+01           2
   0.9000E+00  0.1809E+01           2
   0.1000E+01  0.1100E+01           2
   0.1100E+01  0.6687E+00           2
   0.1200E+01  0.4065E+00           2
   0.1300E+01  0.2472E+00           2
   0.1400E+01  0.1503E+00           2
   0.1500E+01  0.9135E-01           2
```

Figure 8.13 Results from Program 8.3 example

Program 8.4 Plane or axisymmetric analysis of the consolidation equation using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$- or $y(z)$-direction. Explicit time integration using the "theta = 0" method.

```
PROGRAM p84
!-------------------------------------------------------------------------
! Program 8.4 Plane or axisymmetric analysis of the consolidation equation
!             using 4-node rectangular quadrilaterals. Mesh numbered in
!             x(r)- or y(z)- direction. Explicit time integration using
!             the "theta=0" method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,j,nci,ndim=2,nels,neq,nip=4,nn,nod=4,npri, &
   np_types,nres,nstep,ntime,nxe,nye
 REAL(iwp)::det,dtim,one=1.0_iwp,time,zero=0.0_iwp
 CHARACTER(len=15)::dir,element='quadrilateral',type_2d
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),der(:,:),deriv(:,:),fun(:),gc(:),      &
   globma(:),g_coord(:,:),jac(:,:),kay(:,:),kc(:,:),loads(:),mass(:),     &
   newlo(:),ntn(:,:),mm(:,:),points(:,:),prop(:,:),store_mm(:,:,:),       &
   value(:),weights(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)type_2d,dir,nxe,nye,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
 WRITE(11,'(a,i5,a)')" There are",neq," equations"
 ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),globma(0:neq),     &
   coord(nod,ndim),fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),&
   deriv(ndim,nod),mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),  &
   num(nod),etype(nels),x_coords(nxe+1),y_coords(nye+1),                  &
   prop(ndim,np_types),gc(ndim),store_mm(nod,nod,nels),mass(nod),         &
   loads(0:neq),newlo(0:neq))
 READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,npri,nres,ntime
 CALL sample(element,points,weights); globma=zero; gc=one
!---------create and store element and global lumped mass matrices--------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); kc=zero; mm=zero
   gauss_pts: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); if(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
     kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
     CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
   END DO gauss_pts
   DO i=1,nod; mass(i)=SUM(mm(i,:)); END DO; mm=zero
   DO i=1,nod; mm(i,i)=mass(i); END DO
   store_mm(:,:,iel)=mm-kc*dtim; globma(num)=globma(num)+mass
 END DO elements_1; globma(1:)=one/globma(1:); CALL mesh(g_coord,g_num,12)
!-----------------------specify initial and boundary values---------------
 READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)then
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
 END IF
!----------------------time stepping loop-------------------------------
 WRITE(11,'(/a,i3,a)')"    Time      Pressure (node",nres,")"
 WRITE(11,'(2e12.4)')0.0,loads(nres)
 timesteps: DO j=1,nstep
   time=j*dtim; newlo=zero
   elements_2: DO iel=1,nels
     num=g_num(:,iel); mm=store_mm(:,:,iel)
     newlo(num)=newlo(num)+MATMUL(mm,loads(num))
   END DO elements_2; newlo(0)=zero; loads=newlo*globma
   IF(fixed_freedoms/=0)loads(node)=value
   IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
     CALL contour(loads,g_coord,g_num,nci,13); END IF
   IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
 END DO timesteps
STOP
END PROGRAM p84
```

New dynamic real arrays:

`globma`	global lumped mass matrix and its inverse (stored as a vector)
`mass`	element lumped mass vector
`store_mm`	stores lhs element matrices

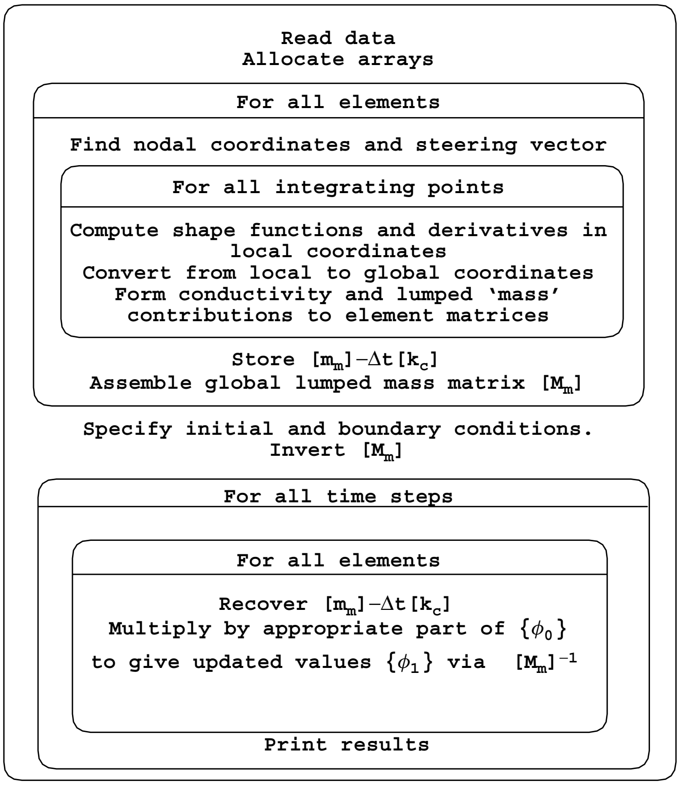

Read data
Allocate arrays
For all elements
Find nodal coordinates and steering vector
For all integrating points
Compute shape functions and derivatives in local coordinates
Convert from local to global coordinates
Form conductivity and lumped 'mass' contributions to element matrices
Store $[m_m]-\Delta t[k_c]$
Assemble global lumped mass matrix $[M_m]$
Specify initial and boundary conditions.
Invert $[M_m]$
For all time steps
For all elements
Recover $[m_m]-\Delta t[k_c]$
Multiply by appropriate part of $\{\phi_0\}$
to give updated values $\{\phi_1\}$ via $[M_m]^{-1}$
Print results

Figure 8.14 Structure chart for explicit time integration from Program 8.4

In non-linear problems which are “ill-conditioned”, pcg iterations could become significant despite the unconditional stability exhibited by Program 8.3. The oldest and simplest mesh-free process using lumped mass, is the “explicit” method, based on equation (3.98), where

$$\{\mathbf{\Phi}\}_1 = [\mathbf{M}_m]^{-1}\,([\mathbf{M}_m] - \Delta t\,[\mathbf{K}_c])\,\{\mathbf{\Phi}\}_0 \tag{8.2}$$

The element integration loop follows the now standard course but the element matrix $([\mathbf{m}_m] - \Delta t[\mathbf{k}_c])$ is stored in `store_mm` for each element rather than assembled as would be the case for a traditional implicit technique. Then in the time-stepping loop, element matrices are recovered from `store_mm` and the product $([\mathbf{M}_m] - \Delta t\,[\mathbf{K}_c])\,\{\Phi\}_0$ formed “element-by-element” using the summation,

$$\sum_{i=1}^{\texttt{nels}} ([\mathbf{m}_m] - \Delta t[\mathbf{k}_c])_i\,\{\boldsymbol{\phi}\}_{0i}$$

where $\{\boldsymbol{\phi}\}_{0i}$ is the appropriate part of $\{\mathbf{\Phi}\}_0$ for element i. The result of this element-by-element product is called `newlo` in programming terminology.

This having been done, the global $\{\mathbf{\Phi}\}_1$ called `loads` is computed by multiplying `newlo` by the inverse of the global mass matrix `globma`. The process is illustrated by the structure chart in Figure 8.14.

The problem shown on Figure 8.5 has been analysed again with the data shown in Figure 8.15 with output as Figure 8.16. The only difference from Figure 8.5, is that being

```
type_2d  dir
'plane'  'x'

nxe  nye  np_types
5    5    1

prop(cx,cy)
1.0  1.0

etype(not needed)

x_coords, y_coords
 0.0  0.2  0.4  0.6  0.8  1.0
 0.0 -0.2 -0.4 -0.6 -0.8 -1.0

dtim   nstep  npri  nres  ntime
0.01   150    10    31    100

loads(i),i=1,neq
  0.0   0.0   0.0   0.0   0.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0
100.0 100.0 100.0 100.0 100.0   0.0


fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
11
 1  0.0    2  0.0    3  0.0    4  0.0    5  0.0    6  0.0
12  0.0   18  0.0   24  0.0   30  0.0   36  0.0

nci
10
```

Figure 8.15 Data for Program 8.4 example

```
 There are   36 equations

    Time       Pressure (node 31)
  0.0000E+00  0.1000E+03
  0.1000E+00  0.8972E+02
  0.2000E+00  0.5881E+02
  0.3000E+00  0.3624E+02
  0.4000E+00  0.2215E+02
  0.5000E+00  0.1353E+02
  0.6000E+00  0.8258E+01
  0.7000E+00  0.5042E+01
  0.8000E+00  0.3078E+01
  0.9000E+00  0.1879E+01
  0.1000E+01  0.1147E+01
  0.1100E+01  0.7005E+00
  0.1200E+01  0.4277E+00
  0.1300E+01  0.2611E+00
  0.1400E+01  0.1594E+00
  0.1500E+01  0.9733E-01
```

Figure 8.16 Results from Program 8.4 example

an "explicit" algorithm, `theta` is automatically set to zero and hence it is no longer required as data.

Program 8.5 Plane or axisymmetric analysis of the consolidation equation using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$- or $y(z)$-direction. "theta" method using an element-by-element product algorithm.

```
PROGRAM p85
!-------------------------------------------------------------------------
! Program 8.5 Plane or axisymmetric analysis of the consolidation equation
!             using 4-node rectangular quadrilaterals. Mesh numbered in
!             x(r)- or y(z)- direction. "theta" method using an
!             element by element (ebe) product algorithm.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,j,nci,ndim=2,nels,neq,nip=4,nn,nod=4,npri, &
   np_types,nres,nstep,ntime,nxe,nye
 REAL(iwp)::det,dtim,one=1.0_iwp,pt5=0.5_iwp,theta,time,zero=0.0_iwp
 CHARACTER(LEN=15)::dir,element='quadrilateral',type_2d
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),der(:,:),deriv(:,:),fun(:),gc(:),      &
   globma(:),g_coord(:,:),jac(:,:),kay(:,:),kc(:,:),loads(:),ntn(:,:),    &
   mm(:,:),points(:,:),prop(:,:),store_kc(:,:,:),value(:),weights(:),     &
   x_coords(:),y_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)type_2d,dir,nxe,nye,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
 WRITE(11,'(a,i5,a)')" There are",neq," equations"
 ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim),   &
   fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),&
   mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod),         &
   globma(0:nn),store_kc(nod,nod,nels),gc(ndim),loads(0:neq),             &
   x_coords(nxe+1),y_coords(nye+1),prop(ndim,np_types),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 READ(10,*)dtim,nstep,theta,npri,nres,ntime
 CALL sample(element,points,weights); globma=zero; gc=one
 !---------create and store element and global lumped mass matrices--------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); kc=zero; mm=zero
   gauss_pts: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); if(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
     kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
     CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
   END DO gauss_pts; store_kc(:,:,iel)=kc
   DO i=1,nod; globma(num(i))=globma(num(i))+SUM(mm(i,:)); END DO
   globma(0)=zero
 END DO elements_1; CALL mesh(g_coord,g_num,12)
!-----------------------recover element A and B matrices------------------
 elements_2: DO iel=1,nels
   num=g_num(:,iel); kc=-store_kc(:,:,iel)*(one-theta)*dtim*pt5
   mm=store_kc(:,:,iel)*theta*dtim*pt5
   DO i=1,nod
     mm(i,i)=mm(i,i)+globma(num(i)); kc(i,i)=kc(i,i)+globma(num(i))
   END DO; CALL invert(mm); mm=MATMUL(mm,kc); store_kc(:,:,iel)=mm
 END DO elements_2
!-----------------------specify initial and boundary values---------------
 READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)THEN
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
 END IF
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/a,i3,a)')"    Time      Pressure (node",nres,")"
 WRITE(11,'(2e12.4)')0.0,loads(nres)
 timesteps: DO j=1,nstep
   time=j*dtim
!-----------------------first pass (1 to nels)----------------------------
   elements_3: DO iel=1,nels
     num=g_num(:,iel); mm=store_kc(:,:,iel)
     loads(num)=MATMUL(mm,loads(num)); loads(0)=zero; loads(node)=value
   END DO elements_3
!-----------------------second pass (nels to 1)---------------------------
   elements_4: DO iel=nels,1,-1
     num=g_num(:,iel); mm=store_kc(:,:,iel)
     loads(num)=MATMUL(mm,loads(num)); loads(0)=zero; loads(node)=value
   END DO elements_4
   IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
     CALL contour(loads,g_coord,g_num,nci,13); END IF
   IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
 END DO timesteps
STOP
END PROGRAM p85
```

New dynamic real arrays:

`store_kc`	stores element `kc` matrices

```
There are   36 equations

    Time       Pressure (node 31)
  0.0000E+00  0.1000E+03
  0.1000E+00  0.8912E+02
  0.2000E+00  0.6107E+02
  0.3000E+00  0.3894E+02
  0.4000E+00  0.2450E+02
  0.5000E+00  0.1537E+02
  0.6000E+00  0.9644E+01
  0.7000E+00  0.6050E+01
  0.8000E+00  0.3795E+01
  0.9000E+00  0.2380E+01
  0.1000E+01  0.1493E+01
  0.1100E+01  0.9366E+00
  0.1200E+01  0.5875E+00
  0.1300E+01  0.3685E+00
  0.1400E+01  0.2312E+00
  0.1500E+01  0.1450E+00
```

Figure 8.17 Results from Program 8.5 example

The motivation in using this algorithm is to preserve the storage economy achieved by the previous explicit technique while attaining the stability properties enjoyed by implicit methods typified by Programs 8.1, 8.2, and 8.3 without involving solution of sets of global equations. The process is described in Chapter 3 by equation (3.101) and in structure by Figure 3.20. The program bears a resemblance to Program 8.4. The element integration loop is employed to store the element matrices $[\mathbf{k}_c]$ in a storage array `store_kc`. In addition, the element consistent mass matrices held in `mm` are diagonalised and the global mass vector, `globma`, assembled. A second loop over the elements is then made, headed "recover element matrices". These are the matrices given in Figure 3.20 by $[\mathbf{m}_m] - (1 - \theta)\Delta t[\mathbf{k}_c]/2$ and $[\mathbf{m}_m] + \theta\Delta t[\mathbf{k}_c]/2$, and are called `kc` and `mm` respectively in the program. The algorithm calls for $[\mathbf{B}]$ (`mm`) to be inverted, which is done using the library subroutine `invert`. Then $[\mathbf{A}]$ is formed as $[\mathbf{B}]^{-1}\left[[\mathbf{m}_m] - (1 - \theta)\Delta t[\mathbf{k}_c]/2\right]$, by multiplying `mm` and `kc`. The result, called `mm` in the program, is re-stored as `store_kc`.

Initial conditions can then be prescribed and the time-stepping loop entered. Within that loop, two passes are made over the elements from first to last and back again. Half of the total $\Delta t[\mathbf{k}_c]$ increment, operates on each pass, and this has been accounted for already in forming $[\mathbf{A}]$ and $[\mathbf{B}]$. The essential coding recovers each element $[\mathbf{B}]^{-1}[\mathbf{A}]$ matrix from `store_kc` and multiplies it by the appropriate part of the solution `loads`. Note that in this product algorithm the solution is continually being updated so there is no need for any "new loads" vector such as had to be employed in the explicit summation algorithm.

The data for this example are identical to those given in Figure 8.5, with results given in Figure 8.17.

8.3 Comparison of Programs 8.2, 8.3, 8.4, and 8.5

These four programs already described in this chapter can all be used to solve plane or axisymmetric conduction or uncoupled consolidation problems. Comparison of Figures 8.10, 8.13, 8.16, and 8.17 shows that for the chosen problem—at the time step used (that is 0.01)—all solutions are accurate, and indeed the explicit solution (Figure 8.16) is probably as accurate as any despite being the simplest and cheapest to obtain.

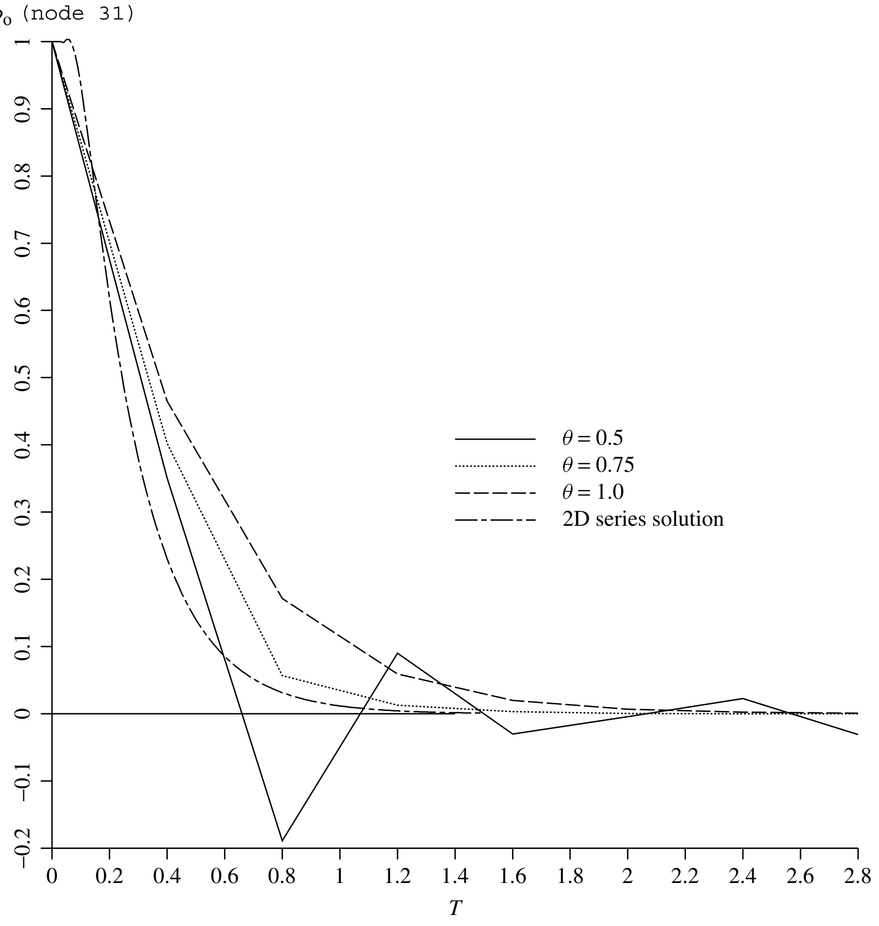


Figure 8.18 Typical solutions from Program 8.2 with varying θ ($\Delta t = 0.4$)

It must, however, be remembered that as the time step Δt is increased, the explicit algorithm will lead to unstable results (the stability limit for the chosen problem is about $\Delta t_{\text{crit}} = 0.02$). At that time step, Program 8.2 with $\theta = 0.5$ will tend to produce oscillatory results, which can be damped, at the expense of average accuracy, by increasing θ towards 1.0. Typical behaviour of the implicit algorithm is illustrated in Figure 8.18.

Program 8.5, while retaining the storage economies of Program 8.4, allows the time step to be increased well beyond the explicit limit. For example, in the selected problem, reasonable results are still produced at $\Delta t = 10\Delta t_{\text{crit}}$. However, as Δt is increased still further, accuracy becomes poorer and Program 8.2 yields the best solutions for very large Δt.

It will be clear that algorithm choice in this area is not a simple one and depends on the nature of the problem (degree of non-linearity, etc.) and on the hardware employed. All the mesh-free methods afford much scope for parallelisation and this is exploited in Chapter 12.

Program 8.6 General two- (plane) or three-dimensional analysis of the consolidation equation. Implicit time integration using the "theta" method.

```
PROGRAM p86
!-------------------------------------------------------------------------
! Program 8.6 General two- (plane) or three-dimensional analysis of the
!             consolidation equation. Implicit time integration using
!             the "theta" method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,j,nci,ndim,nels,neq,nip,nn,nod=4,npri,     &
   np_types,nres,nstep,ntime
 REAL(iwp)::det,dtim,penalty=1.0e20_iwp,theta,time,zero=0.0_iwp
 CHARACTER(len=15)::element
!---------------------------- dynamic arrays----------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),node(:),num(:),kdiag(:)
 REAL(iwp),ALLOCATABLE::bp(:),coord(:,:),der(:,:),deriv(:,:),fun(:),      &
   g_coord(:,:),jac(:,:),kay(:,:),kc(:,:),kv(:),loads(:),newlo(:),        &
   ntn(:,:),mm(:,:),points(:,:),prop(:,:),storbp(:),value(:),weights(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)element,nels,nn,nip,nod,ndim,np_types; neq=nn
 ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),fun(nod),     &
   etype(nels),jac(ndim,ndim),weights(nip),num(nod),ntn(nod,nod),         &
   g_num(nod,nels),der(ndim,nod),deriv(ndim,nod),kc(nod,nod),mm(nod,nod), &
   kay(ndim,ndim),kdiag(neq),prop(ndim,np_types),newlo(0:neq),loads(0:neq))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)g_coord; READ(10,*)g_num; READ(10,*)dtim,nstep,theta,npri,nres
 IF(ndim==2.AND.nod==4)THEN
   READ(10,*)ntime,nci; CALL mesh(g_coord,g_num,12)
 END IF; kdiag=0
!-----------loop the elements to set up global geometry and kdiag --------
 elements_1: DO iel=1,nels
   num=g_num(:,iel); CALL fkdiag(kdiag,num)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 WRITE(11,'(2(a,i5))')                                                    &
 " There are",neq,"  equations and the skyline storage is ",kdiag(neq)
 ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq)))
 CALL sample(element,points,weights); kv=zero; bp=zero
!------------- global conductivity matrix assembly-----------------------
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
   gauss_pts: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der)
     kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
     CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)
   END DO gauss_pts
   CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
 END DO elements_2; kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta
!---------------initial and boundary conditions data---------------------
 READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
 IF(fixed_freedoms/=0)then
   ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),                   &
          storbp(fixed_freedoms))
   READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
   bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
 END IF
!-----------------------factorise left hand side------------------------
 CALL sparin(bp,kdiag)
!------------------time stepping recursion------------------------------
 WRITE(11,'(/a,i3,a)')"    Time      Pressure (node",nres,")"
 WRITE(11,'(2e12.4)')0.0,loads(nres)
 timesteps: DO j=1,nstep
   time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag)
   IF(fixed_freedoms/=0)newlo(node)=storbp*value
   CALL spabac(bp,newlo,kdiag); loads=newlo
   IF(ndim==2.AND.nod==4.AND.j==ntime)                                  &
     CALL contour(loads,g_coord,g_num,nci,13)
   IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
 END DO timesteps
STOP
END PROGRAM p86
```

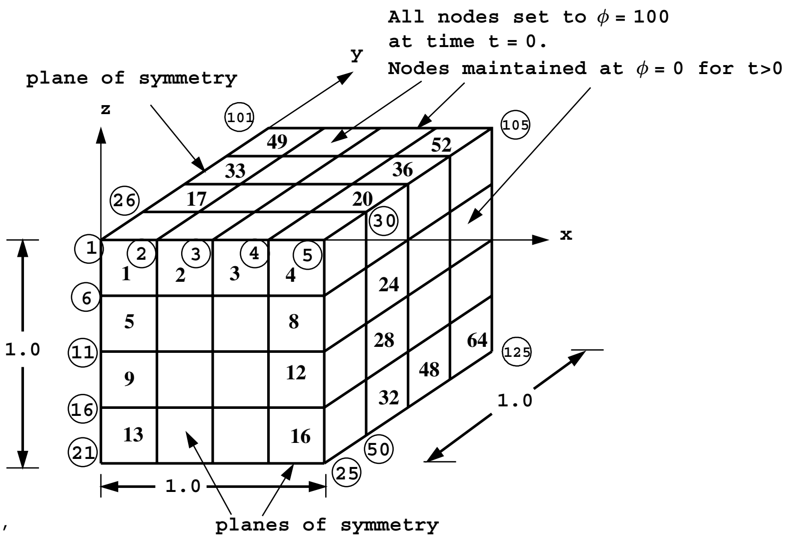
Program 8.6 is the last of the diffusion or "consolidation" programs in this chapter and the most general. The program can analyse consolidation over any two- or three-dimensional domain with non-homogeneous and anisotropic material properties. The program is very similar to Program 5.5 for general elastic analysis and Program 7.4 for general steady seepage analysis. As with those programs, a variety of 2D or 3D elements can be chosen through the data file. The graphics subroutines `mesh` and `contour` are only activated for 2D analysis, and in the current versions, contouring is only available when using 4-node quadrilateral elements.

The main difference from earlier programs in the chapter (e.g. Program 8.2), is that this program includes no "geometry" subroutine, so all nodal coordinates (`g_coords`) and element node numbers (`g_num`) must be provided as data. In addition, some of the variables that were previously fixed in the declaration statements, must now be read as data in order to identify the dimensionality of the problem and the type of element required. There are no variables required by this program that have not already been encountered in earlier programs of this chapter.

A three-dimensional consolidation example is shown in Figure 8.19. The model represents one-eighth of a symmetrical cube with drainage permitted at all its outer surfaces. The finite element model uses 8-node hexahedral elements, and includes 64 elements and 125 nodes. The initial condition is that the excess pore pressure at all nodes is set equal to 100.0. The objective is to compute the excess pore pressure at the centre of the cube as time passes.

The first line of data identifies the element type, which in this case is a `'hexahedron'`. The next line gives the number of elements `nels`, the number of nodes `nn`, the number of integrating points `nip`, the number of nodes on each element `nod`, the number of dimensions of the problem `ndim` and the number of property types `np_types`. It may also be noted that numerical integration of an 8-node hexahedral element usually involves 8 Gauss points, (2 in each of the three coordinate directions), so `nip` is read as 8.

The problem is homogeneous and isotropic, so the next line indicates that the soil has coefficients of consolidation equal to unity in all three coordinate directions ($c_x = c_y = c_z = 1.0$). With `np_types=1`, no `etype` data is required.



```
element
'hexahedron'

nels   nn   nip   nod   ndim   np_types
64    125    8     8     3       1

prop(cx,cy,cz)
1.0  1.0  1.0

etype(not needed)

g_coord
0.00  0.00  0.00   0.25  0.00  0.00   0.50  0.00  0.00   0.75  0.00  0.00
g_coord data for nodes 5-120 omitted here
0.00  1.00 -1.00   0.25  1.00 -1.00   0.50  1.00 -1.00   0.75  1.00 -1.00
1.00  1.00 -1.00

g_num
  6   1   2   7  31  26  27  32      7   2   3   8  32  27  28  33
g_num data for elements 3-62 omitted here
 98  93  94  99 123 118 119 124     99  94  95 100 124 119 120 125

dtim  nstep  theta  npri  nres
0.01   150    0.5    10    21

loads(i),i=1,neq
Top, right and back faces set to zero. All other nodes set to 100.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
 61
  1 0.0    2 0.0    3 0.0    4 0.0    5 0.0   10 0.0   15 0.0   20 0.0
fixed freedom data for 48 nodes omitted here
121 0.0  122 0.0  123 0.0  124 0.0  125 0.0
```

Figure 8.19 Mesh and data for Program 8.6 example

The next data involves the 125 (x, y, z) coordinates of the mesh read into g_coord, followed by the 64 groups of 8 node numbers attached to each element read into g_num. If dealing with a three-dimensional 8-node element for example, the order in which the node numbers are read must follow the ordering described in Appendix B. Due to the volume of data required in this example, a truncated version of the data is actually shown in Figure 8.19.

```
There are  125  equations and the skyline storage is  3225

   Time       Pressure (node 21)
 0.0000E+00  0.1000E+03
 0.1000E+00  0.8528E+02
 0.2000E+00  0.4386E+02
 0.3000E+00  0.2090E+02
 0.4000E+00  0.9878E+01
 0.5000E+00  0.4666E+01
 0.6000E+00  0.2204E+01
 0.7000E+00  0.1041E+01
 0.8000E+00  0.4916E+00
 0.9000E+00  0.2322E+00
 0.1000E+01  0.1097E+00
 0.1100E+01  0.5179E-01
 0.1200E+01  0.2446E-01
 0.1300E+01  0.1155E-01
 0.1400E+01  0.5457E-02
 0.1500E+01  0.2577E-02
```

Figure 8.20 Results from the Program 8.6 example

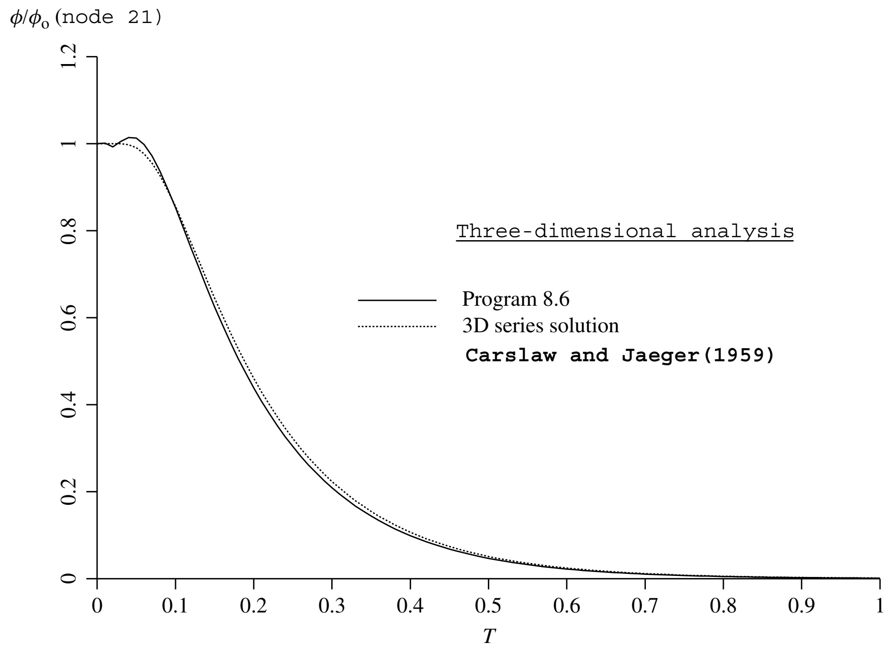


Figure 8.21 Comparison of Program 8.6 result for a cube with 3D series solution

The time-stepping and output data follows next, and involve the usual parameters. Output is requested at node `nres=21`. It should be noted that `ntime` and `nci` are not read in this case because there is no contouring option in 3D.

The output from the program is shown in Figure 8.20, and gives the rate of pore pressure dissipation at the centre of the cube. Figure 8.21 shows a plot of this result compared with the 3D series solution from Carslaw and Jaeger (1959).

Program 8.7 Plane analysis of the diffusion–convection equation using 4-node rectangular quadrilaterals. Implicit time integration using the "theta" method. Self-adjoint transformation.

```
PROGRAM p87
!-------------------------------------------------------------------------
! Program 8.7 Plane analysis of the diffusion-convection equation
!             using 4-node rectangular quadrilaterals. Implicit time
!             integration using the "theta" method.
!             Self-adjoint transformation.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,j,nci,ndim=2,nels,neq,nip=4,nn,nod=4,npri,np_types,nres,  &
   nstep,ntime,nxe,nye
 REAL(iwp)::det,dtim,d6=6.0_iwp,d12=12.0_iwp,f1,f2,pt25=0.25_iwp,theta,   &
   time,two=2.0_iwp,ux,uy,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),num(:),kdiag(:)
 REAL(iwp),ALLOCATABLE::ans(:),bp(:),coord(:,:),der(:,:),deriv(:,:),      &
   fun(:),g_coord(:,:),jac(:,:),kay(:,:),kc(:,:),kv(:),loads(:),ntn(:,:), &
   mm(:,:),points(:,:),prop(:,:),weights(:),x_coords(:),y_coords(:)
!-------------------------input and initialisation-----------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
 ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim),   &
   fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),&
   mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod),         &
   prop(ndim,np_types),x_coords(nxe+1),y_coords(nye+1),etype(nels),       &
   kdiag(neq),loads(0:neq),ans(0:neq))
 READ(10,*)prop; etype=1; if(np_types>1)read(10,*)etype
 READ(10,*)x_coords,y_coords
 READ(10,*)dtim,nstep,theta,npri,nres,ntime,ux,uy,nci; kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
   CALL fkdiag(kdiag,num)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq)))
 WRITE(11,'(2(a,i5),/)')                                                  &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); kv=zero; bp=zero
!-----------------------global conductivity and "mass" matrix assembly----
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
   gauss_pts: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der)
     kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
     CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)
   END DO gauss_pts
   kc=kc+mm*(ux*ux/kay(1,1)+uy*uy/kay(2,2))*pt25; mm=mm/(theta*dtim)
```

```
!-----------------------derivative boundary conditions--------------------
   IF(iel==1)THEN
     det=x_coords(2)-x_coords(1)
     kc(2,2)=kc(2,2)+uy*det/d6; kc(2,3)=kc(2,3)+uy*det/d12
     kc(3,2)=kc(3,2)+uy*det/d12; kc(3,3)=kc(3,3)+uy*det/d6
   ELSE IF(iel==nels)THEN
     det=x_coords(2)-x_coords(1)
     kc(1,1)=kc(1,1)+uy*det/d6; kc(1,4)=kc(1,4)+uy*det/d12
     kc(4,1)=kc(4,1)+uy*det/d12; kc(4,4)=kc(4,4)+uy*det/d6
   END IF; CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
 END DO elements_2;f1=uy*det/(two*theta); f2=f1; bp=bp+kv; kv=bp-kv/theta
!-----------------------factorise equations------------------------------
 CALL sparin(bp,kdiag); loads=zero
!-----------------------time stepping loop-------------------------------
 WRITE(11,'(a,i3,a)')"    Time     Concentration (node",nres,")"
 WRITE(11,'(2e12.4)')0.0,loads(nres)
 timesteps: DO j=1,nstep
   time=j*dtim; CALL linmul_sky(kv,loads,ans,kdiag)
   ans(neq)=ans(neq)+f1; ans(neq-1)=ans(neq-1)+f2
   CALL spabac(bp,ans,kdiag); loads=ans
   IF(nod==4.AND.j==ntime)CALL contour(loads,g_coord,g_num,nci,13)
   IF(j/npri*npri==j)WRITE(11,'(2e12.4)')                                &
     time,loads(nres)*exp(-ux*g_coord(1,nres)/two/kay(1,1))*             &
     exp(-uy*g_coord(2,nres)/two/kay(2,2))
 END DO timesteps
STOP
END PROGRAM p87
```

New scalar reals:

`d6`	set to 6.0
`d12`	set to 12.0
`f1`	used to fix derivative boundary condition
`f2`	used to fix derivative boundary condition
`pt25`	set to 0.25
`ux`	velocity in x-direction
`uy`	velocity in y-direction

New dynamic real arrays:

`ans`	rhs vector in equilibrium equations

When convection terms are retained in the simplified flow equations, (2.132) has to be solved. Again many techniques could be employed but, in the present program, an implicit algorithm based on equation (3.94) is used. Thus this program is an extension of Program 8.2.

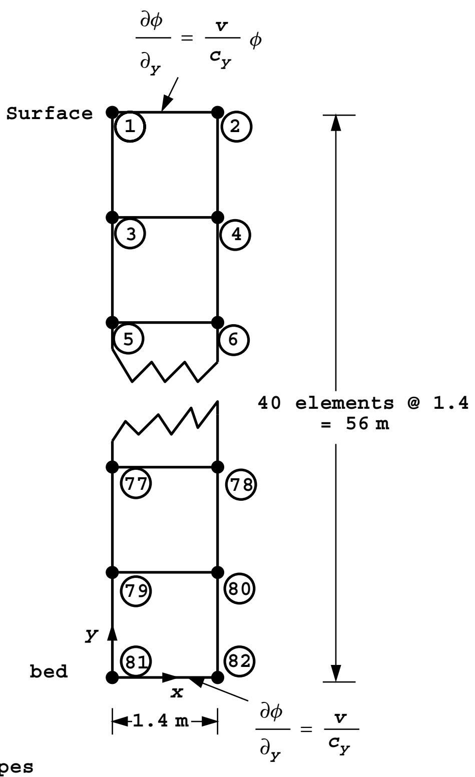
When the transformation of equation (2.134) is employed, the equation to be solved becomes,

$$c_x \frac{\partial^2 h}{\partial x^2} + c_y \frac{\partial^2 h}{\partial y^2} - \left(\frac{u^2}{4\,c_x} + \frac{v^2}{4\,c_y} \right) h = \frac{\partial h}{\partial t} \tag{8.3}$$

thus the extra term involving h distinguishes the process from a simple diffusion one. However, reference to Table 2.1 shows that the semi-discretised "stiffness" matrix for this problem will still be symmetrical, the h term involving an element matrix of the "mass matrix" type, namely $\int N_i N_j \, \mathrm{d}x \, \mathrm{d}y$.

Comparison with Program 8.2 will show essentially the same array declarations and input parameters. Extra variables are the velocities u and v in the x and y directions, `ux` and `uy` respectively.

The problem chosen is the one-dimensional example shown in Figure 8.22 consisting of a 56-m deep bed of fluid discretised by 40, 4-noded elements. There is a steady velocity in



$\frac{\partial \phi}{\partial_y} = \frac{v}{c_y}\phi$
Surface
1
2
3
4
5
6
40 elements @ 1.4
= 56 m
77
78
79
80
y
81
82
bed
x
1.4 m
$\frac{\partial \phi}{\partial_y} = \frac{v}{c_y}$

```
nxe   nye   np_types
1     40      1

prop(cx,cy)
1.0e-6  0.49

etype(not needed)

x_coords  y_coords
  0.0     1.4
 56.0    54.6    53.2    51.8    50.4    49.0    47.6    46.2    44.8    43.4
 42.0    40.6    39.2    37.8    36.4    35.0    33.6    32.2    30.8    29.4
 28.0    26.6    25.2    23.8    22.4    21.0    19.6    18.2    16.8    15.4
 14.0    12.6    11.2     9.8     8.4     7.0     5.6     4.2     2.8     1.4
  0.0

dtim   nstep   theta   npri   nres   ntime   ux     uy
300.0  20       1.0     1      82      5     0.0  0.0135

nci
10
```

Figure 8.22 Mesh and data for Program 8.7 example

the y-direction (uy) of 0.0135, and the initial concentration at all points is set to zero. The dependent variable ϕ refers to the concentration of sediment picked up by the flow from the base of the mesh ($y = 0.0$), and distributed with time in the y-direction. The velocity in the x-direction (ux) is zero, and for numerical reasons c_x is set to a small number, 1×10^{-6}, which is effectively zero. After reading the mesh coordinate data `x_coords` and `y_coords`, the implicit time-stepping parameters `dtim`, `nsteps` and `theta` are read, followed by the output parameters `npri`, `nres` and `ntime`. In this example, a fully "implicit" time-stepping scheme is illustrated by setting `theta=1`. The output calls for results to be printed every time step at node 82. The steady velocities are read as `ux` and `uy` and the contour map of concentration after `ntime` time steps will be written to file `fe95.con` with `nci` contour intervals.

The equation to be solved reduces to,

$$c_y \frac{\partial^2 h}{\partial y^2} - \frac{v^2}{4c_y} h = \frac{\partial h}{\partial t} \tag{8.4}$$

subject to the boundary conditions at $y = 0$ of

$$\frac{\partial \phi}{\partial y} = \frac{v}{c_y} = C_2 \tag{8.5}$$

and at $y = 56.0$ of

$$\frac{\partial \phi}{\partial y} = \frac{v}{c_y} \phi = C_2 \phi \tag{8.6}$$

After transformation, these conditions become

$$\frac{\partial h}{\partial y} = -\frac{v}{2c_y} h + \frac{v}{c_y} \tag{8.7}$$

and

$$\frac{\partial h}{\partial y} = \frac{v}{2c_y} h \tag{8.8}$$

Boundary condition (8.8) is clearly of the type described in Section 3.6, equation (3.33), hence at that boundary, the element matrix will have to be augmented by the matrix shown in equation (3.37). The multiple $C_1 c_y (x_k - x_j)/6$ in (3.37) is just $v(x_k - x_j)/12$ or `uy*(x_coords(2)-x_coords(1))/12` in the program. This is carried out in the section of program headed "derivative boundary conditions".

The condition (8.7) contains a similar contribution, but in addition the term v/c_y is of the type described by the equation (3.34). Thus, an addition must be made to the right-hand side of the equations at such a boundary in accordance with equation (3.39). In this case the terms in equation (3.39) are $v(x_k - x_j)/2$ and are incorporated in the program immediately after the comment "factorise equations".

This example shows that quite complicated coding would be necessary to permit very general boundary conditions to be specified in all problems. The authors prefer to write specialised code when necessary.

```
There are   82 equations and the skyline storage is  283

   Time      Concentration (node 82)
 0.0000E+00  0.0000E+00
 0.3000E+03  0.2828E+00
 0.6000E+03  0.4011E+00
 0.9000E+03  0.4796E+00
 0.1200E+04  0.5389E+00
 0.1500E+04  0.5867E+00
 0.1800E+04  0.6269E+00
 0.2100E+04  0.6616E+00
 0.2400E+04  0.6920E+00
 0.2700E+04  0.7192E+00
 0.3000E+04  0.7435E+00
 0.3300E+04  0.7656E+00
 0.3600E+04  0.7856E+00
 0.3900E+04  0.8038E+00
 0.4200E+04  0.8204E+00
 0.4500E+04  0.8356E+00
 0.4800E+04  0.8494E+00
 0.5100E+04  0.8621E+00
 0.5400E+04  0.8737E+00
 0.5700E+04  0.8843E+00
 0.6000E+04  0.8941E+00
```

Figure 8.23 Results from the Program 8.7 example

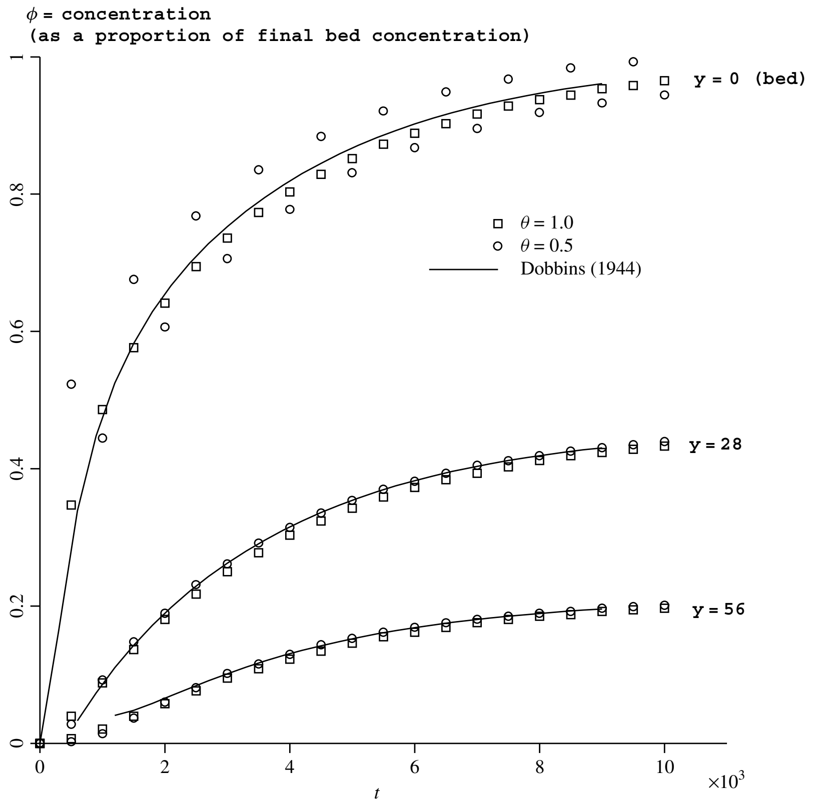


Figure 8.24 Graph of concentration versus time from Program 8.7 example

After insertion of boundary conditions the (constant) global left hand side matrix (bp) is factorised using sparin. The time-stepping loop follows a familiar course, with matrix-by-vector multiplication using subroutine linmul_sky followed by forward and backsubstitution using spabac. Output for twenty steps is listed in Figure 8.23, while Figure 8.24 shows how the finite element solution compares with an "analytical" one due to Dobbins (1944).

It should be remembered that solutions are in terms of the transformed variable h, and the true solution ϕ has been recovered using (2.134).

Program 8.8 Plane analysis of the diffusion–convection equation using 4-node rectangular quadrilaterals. Implicit time integration using the "theta" method. Untransformed solution.

```
PROGRAM p88
!-------------------------------------------------------------------------
! Program 8.8 Plane analysis of the diffusion-convection equation
!             using 4-node rectangular quadrilaterals. Implicit time
!             integration using the "theta" method.
!             Untransformed solution.
!-------------------------------------------------------------------------
 USE main;  USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,j,k,nband,ndim=2,nels,neq,fixed_freedoms,nip=4,nn,nod=4,  &
   npri,np_types,nres,nstep,ntime,nxe,nye
 REAL(iwp)::det,dtim,part1,part2,pt2=0.2_iwp,penalty=1.0e20_iwp,theta,    &
   time,ux,uy,zero=0.0_iwp; CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g_num(:,:),node(:),num(:)
 REAL(iwp),ALLOCATABLE::ans(:),conc(:),coord(:,:),copy(:,:),der(:,:),     &
   deriv(:,:),dtkd(:,:),fun(:),g_coord(:,:),jac(:,:),kb(:,:),kc(:,:),     &
   loads(:),ntn(:,:),pb(:,:),mm(:,:),points(:,:),prop(:,:),storpb(:),     &
   weights(:),work(:,:),x_coords(:),y_coords(:)
!-------------------------input and initialisation-----------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
 ALLOCATE(points(nip,ndim),weights(nip),coord(nod,ndim),fun(nod),         &
   jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),         &
   mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod),         &
   dtkd(nod,nod),prop(ndim,np_types),x_coords(nxe+1),y_coords(nye+1),     &
   etype(nels),conc(nye+1))
 READ(10,*)prop; etype=1; if(np_types>1)read(10,*)etype
 READ(10,*)x_coords,y_coords
 READ(10,*)dtim,nstep,theta,npri,nres,ntime,ux,uy;  nband=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
   IF(nband<bandwidth(num))nband=bandwidth(num)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 ALLOCATE(kb(neq,2*nband+1),pb(neq,2*nband+1),work(nband+1,neq),          &
          copy(nband+1,neq),loads(0:neq),ans(0:neq))
 WRITE(11,'(2(a,i5))')                                                    &
 " There are",neq," equations and the half-bandwidth is",nband
 CALL sample(element,points,weights); kb=zero; pb=zero
```

```
!-----------------------global conductivity and "mass" matrix assembly----
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
   gauss_pts: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der)
     DO j=1,nod; DO k=1,nod
         part1=prop(1,etype(iel))*deriv(1,j)*deriv(1,k)+                &
           prop(2,etype(iel))*deriv(2,j)*deriv(2,k)
         part2=ux*fun(j)*deriv(1,k)+uy*fun(j)*deriv(2,k)
         dtkd(j,k)=(part1-part2)*det*weights(i)
       END DO; END DO
     kc=kc+dtkd; CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)
   END DO gauss_pts; mm=mm/(theta*dtim)
   CALL formtb(kb,kc,num); CALL formtb(pb,mm,num)
 END DO elements_2; pb=pb+kb; kb=pb-kb/theta
!-----------------------boundary conditions-------------------------------
 READ(10,*)fixed_freedoms
 allocate(node(fixed_freedoms),storpb(fixed_freedoms))
 READ(10,*)node
 pb(node,nband+1)=pb(node,nband+1)+penalty
 storpb=pb(node,nband+1)
!-----------------------factorise equations-------------------------------
 work=zero; CALL gauss_band(pb,work)
 WRITE(11,'(/a,i3,a)')"    Time      Concentration(node",nres,")"
 WRITE(11,'(2e12.4)')0.0,loads(nres); loads=zero
!-----------------------time stepping loop--------------------------------
 timesteps: DO j=1,nstep
   time=j*dtim; copy=work; CALL bantmul(kb,loads,ans); ans(0)=zero
   IF(time<=pt2)THEN; ans(node)=storpb; ELSE; ans(node)=zero; END IF
   CALL solve_band(pb,copy,ans); ans(0)=zero; loads=ans
   IF(j==ntime)conc(:)=loads(2:neq:2)
   IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
 END DO timesteps
 WRITE(11,'(/a,e10.4,a)')                                               &
       "  Distance    Concentration(time=",nstep*dtim,")"
 DO i=1,nye+1; WRITE(11,'(2e12.4)')y_coords(i),conc(i); END DO
STOP
END PROGRAM p88
```

New scalar integers:

`nband`	fixed bandwidth of non-symmetric matrix

New scalar reals:

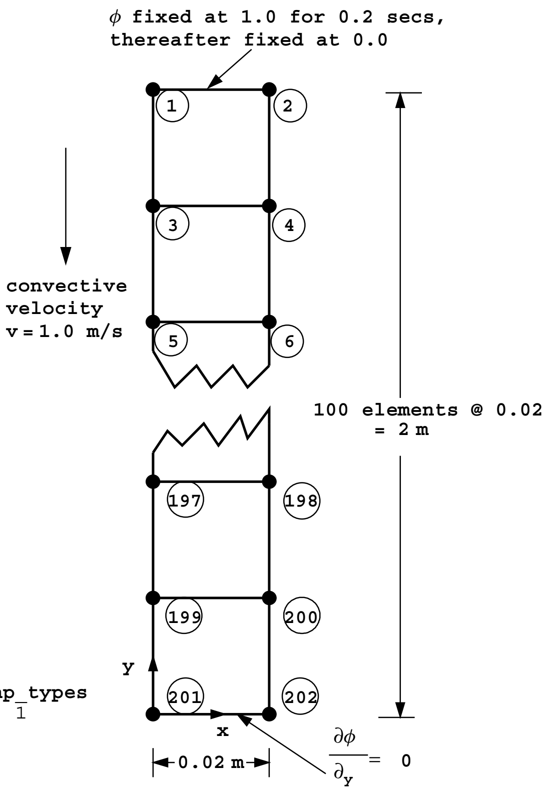
`part1`	diffusive part
`part2`	convective part
`pt2`	set to 0.2

New dynamic real arrays:

`conc`	concentration distribution after `ntime` steps
`copy`	working space array
`storpb`	stores augmented diagonal terms
`work`	working space array

In the previous example, difficulties would have arisen had the ratio of u to c_x been large, because the transformation involving $\exp(u/c_x)$ would not have been numerically feasible (Smith *et al.*, 1973). Under these circumstances (and even when $c_x = c_y = 0$) equation (2.132) can still be solved, but with the drawback that the element and system matrices become unsymmetrical, although the latter are still banded.

Program 8.8 will be used to solve a purely convective problem, that is with $c_x = c_y = 0$. The problem chosen is again one-dimensional as shown in Figure 8.25, so the equation


ϕ fixed at 1.0 for 0.2 secs, thereafter fixed at 0.0
1
2
3
4
5
6
convective velocity v = 1.0 m/s
100 elements @ 0.02 = 2 m
197
198
199
200
201
202
y
x
0.02 m
$\frac{\partial\phi}{\partial y} = 0$

```
nxe   nye   np_types
1     100     1

prop(cx,cy)
 0.0   0.0

etype(not needed)

x_coords, y_coords
 0.00    0.02
 0.00   -0.02   -0.04   -0.06   -0.08   -0.10   -0.12   -0.14   -0.16   -0.18
 .
y_coords(11-90) data omitted here
 .
-1.80   -1.82   -1.84   -1.86   -1.88   -1.90   -1.92   -1.94   -1.96   -1.98
-2.00

dtim   nstep   theta   npri   nres   ntime   ux    uy
0.04    25      0.5     1      3      25     0.0   1.0

fixed_freedoms,(node(i),i=1,fixed_freedoms)
2
1  2
```

Figure 8.25 Mesh and data for Program 8.8 example

is effectively,

$$-v\frac{\partial \phi}{\partial y} = \frac{\partial \phi}{\partial t} \tag{8.9}$$

in the region $0 \leq y \leq 2$, subject to the boundary conditions $\phi = 1$ at $y = 0$ for $0 \leq t \leq 0.2$, $\phi = 0$ at $y = 0$ for $t > 0.2$ and $\partial\phi/\partial y = 0$ at $y = 2$ for all t.

Comparing with Program 8.7, the usual geometry subroutine for 4-noded elements numbered in the x-direction, geom_rect with 'dir = 'x' is used. Arrays node and storpb are used to read in the numbers of fixed freedoms and to store information about them during the time-stepping process. The system kb matrix is now unsymmetrical and stored as a rectangular array with the full bandwidth using subroutine formtb (see Figure 3.18). Although pb is symmetrical, it too is stored as a full band to be compatible with kb.

```
There are  202 equations and the half-bandwidth is    3

   Time       Concentration(node  3)
 0.0000E+00  0.0000E+00
 0.4000E-01  0.3660E+00
 0.8000E-01  0.1116E+01
 0.1200E+00  0.1066E+01
 0.1600E+00  0.9285E+00
 0.2000E+00  0.1017E+01
(some results omitted here)
 0.8800E+00 -0.6148E-02
 0.9200E+00 -0.3116E-02
 0.9600E+00  0.8138E-02
 0.1000E+01 -0.4934E-02

 Distance    Concentration(time=0.1000E+01)
 0.0000E+00  0.3289E-24
-0.2000E-01 -0.4934E-02
-0.4000E-01  0.1548E-01
-0.6000E-01 -0.4225E-02
-0.8000E-01 -0.2975E-01
-0.1000E+00  0.3345E-01
(some results omitted here)
-0.7000E+00 -0.9004E-01
-0.7200E+00  0.1104E+00
-0.7400E+00  0.3613E+00
-0.7600E+00  0.6202E+00
-0.7800E+00  0.8459E+00
-0.8000E+00  0.1009E+01
-0.8200E+00  0.1095E+01
-0.8400E+00  0.1105E+01
-0.8600E+00  0.1051E+01
-0.8800E+00  0.9513E+00
-0.9000E+00  0.8246E+00
-0.9200E+00  0.6881E+00
-0.9400E+00  0.5551E+00
-0.9600E+00  0.4344E+00
-0.9800E+00  0.3306E+00
-0.1000E+01  0.2453E+00
-0.1020E+01  0.1779E+00
-0.1040E+01  0.1262E+00
-0.1060E+01  0.8776E-01
-0.1080E+01  0.5989E-01
-0.1100E+01  0.4017E-01
-0.1120E+01  0.2650E-01
(some results omitted here).
-0.1900E+01  0.1688E-11
-0.1920E+01  0.8274E-12
-0.1940E+01  0.4025E-12
-0.1960E+01  0.1973E-12
-0.1980E+01  0.9360E-13
-0.2000E+01  0.4779E-13
```

Figure 8.26 Results from Program 8.8 example

In the element integration and assembly loop, `part1` accumulates the diffusive part of the element "stiffness" and `part2` the convective part. The `part2` contribution to the `kc` matrix is the only one in the present example and it is unsymmetrical (skew-symmetrical in fact).

The structure of the program is modelled on the previous one. The solution routines are `gauss_band` and `solve_band` which use an extra array `work` as working space. Matrix-by-vector multiplication needs the subroutine `bantmul` (see Table 3.5).

In the section "time-stepping loop" it can be seen that the solution at nodes 1 and 2 is held at the value 1.0 for the first 0.2 s of convection and at zero subsequently.

After reading the mesh coordinate data `x_coords` and `y_coords`, the conventional implicit time-stepping parameters `dtim`, `nsteps` and `theta` are read, followed by the output control parameters. The output calls for results to be printed every time step at node 3. No contour map is produced in this case, but instead the output produces the spatial concentration after `ntime=25` time steps. The velocities are read as `ux=0.0` and `uy=1.0`, followed by the number of fixed freedoms `fixed_freedoms`, and the node numbers to be fixed.

The variation of concentration with time at node 3 and the concentration distribution after 1.0 s are shown in the results file in Figure 8.26. Figure 8.27 gives a plot of the

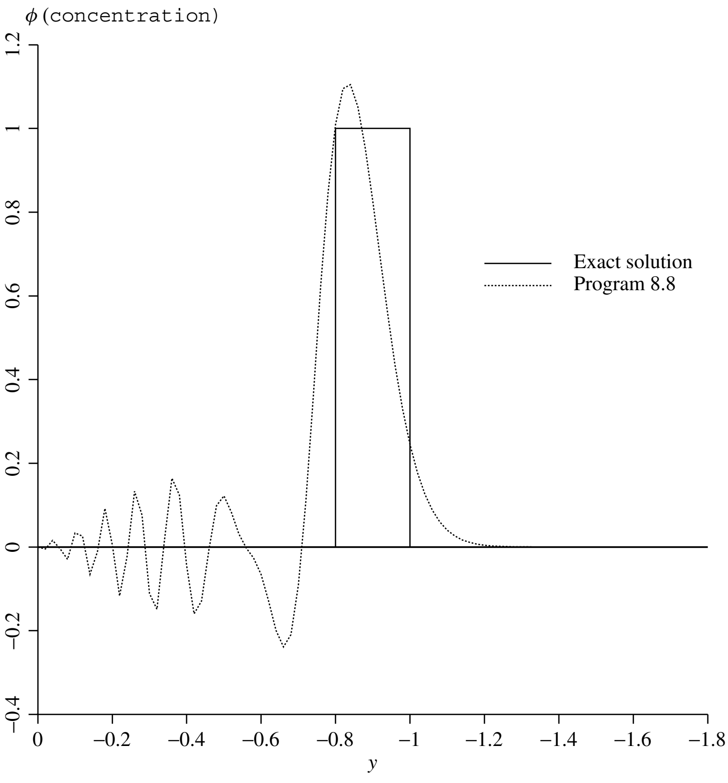


Figure 8.27 Concentration versus distance after 1 s from Program 8.8 example

computed solution after 1 s together with the correct solution to the problem as described by a rectangular pulse moving with unit velocity in the y-direction. Spurious spatial oscillations are seen to have been introduced by the numerical solution. Measures to improve the solutions are beyond the scope of the present treatment (Smith, 1976, 1979). Of course, in the present case improvements can be achieved by simply reducing the element size in the y-direction and the time step size Δt.

Glossary of variable names used in Chapter 8

Scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling
`fixed_freedoms`	number of fixed total heads
`i`	simple counter
`iel`	simple counter
`iwp`	`SELECTED_REAL_KIND(15)`
`j`	simple counter
`k`	simple counter
`nband`	full bandwidth of non-symmetric matrix
`nci`	number of contour intervals
`ndim`	number of dimensions
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integrating points
`nn`	number of nodes in the mesh
`nod`	number of nodes per element
`npri`	output printed every `npri` time steps
`nprops`	number of material properties
`np_types`	number of different property types
`nres`	node number at which time history is to be printed
`nstep`	number of time steps required
`ntime`	time step number at which spatial distribution is to be printed or contoured
`nxe`	number of elements in $x(r)$-direction
`nye`	number of elements in $y(z)$-direction

Scalar reals:

`alpha`	α from equations (3.22)
`at`	holds area beneath isochrone by Trapezoid Rule at time t
`a0`	holds area beneath isochrone by Trapezoid Rule at time $t = 0$
`beta`	β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`det`	determinant of the Jacobian matrix
`dtim`	calculation time step
`d6`	set to 6.0
`d12`	set to 12.0

`f1`	used to fix derivative boundary condition
`f2`	used to fix derivative boundary condition
`one`	set to 1.0
`part1`	diffusive part
`part2`	convective part
`penalty`	set to 1×10^{20}
`pt2`	set to 0.2
`pt25`	set to 0.25
`pt5`	set to 0.5
`theta`	time integration weighting parameter
`time`	holds elapsed time t
`up`	holds dot product $(\mathbf{R}^k)^{\mathrm{T}}(\mathbf{R}^k)$ from equations (3.22)
`ux`	velocity in x-direction
`uy`	velocity in y-direction
`zero`	set to 0.0

Scalar characters:

`dir`	element and node numbering direction
`element`	element type
`type_2d`	type of 2D analysis

Scalar logical:

`cg_converged`	set to `.TRUE.` if pcg process has converged

Dynamic integer arrays:

`etype`	element property types
`g_num`	global element node numbers matrix
`kdiag`	diagonal term locations
`node`	nodes with fixed values
`num`	element node numbers

Dynamic real arrays:

`ans`	rhs vector in equilibrium equations
`bp`	global “mass” matrix
`conc`	concentration distribution after `ntime` steps
`coord`	element nodal coordinates
`copy`	working space array
`d`	vector used in equation (3.22)
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`diag_precon`	diagonal preconditioner vector
`ell`	element lengths
`fun`	shape functions
`gc`	integrating point coordinates
`globma`	global lumped mass matrix (stored as a vector)

`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`kay`	permeability matrix
`kc`	element conductivity matrix
`kv`	global conductivity matrix
`loads`	excess pore pressure values
`mass`	element lumped mass vector
`mm`	element "mass" matrix
`newlo`	new excess pore pressure values
`p`	"descent" vector used in equation (3.22)
`points`	integrating point local coordinates
`press`	excess pore pressure values after `ntime` time steps
`prop`	element properties
`r`	holds fixed rhs terms in pcg solver
`storpb`	stores augmented diagonal terms
`store`	stores global augmented diagonal terms
`storka`	stores lhs element matrix
`storkb`	stores rhs element matrix
`store_kc`	stores element `kc` matrices
`store_mm`	stores lhs element matrices
`u`	vector used in equation (3.22)
`value`	fixed boundary values of excess pore pressure
`weights`	weighting coefficients
`work`	working space array
`x`	"old" solution vector
`xnew`	"new" solution vector
`x_coords`	$x(r)$-coordinates of mesh layout
`y_coords`	$y(z)$-coordinates of mesh layout

8.4 Exercises

1. A layer of clay of thickness $2D$, free draining at its top and bottom surfaces is subjected to a suddenly applied distributed load of one unit. Working in terms of a dimensionless Time Factor given by $T = c_v t/D^2$, and using a single finite element, use the Crank–Nicolson approach ($\theta = 0.5$) with a time step of $\Delta T = 0.1$ to estimate the mid-depth pore pressure when $T = 0.3$. Compare this result with the analytical solution to your equation.
   (Ans: Numerical 0.40; Analytical 0.41)

2. A rod of length 1 unit and thermal diffusivity 1 unit is initially at a temperature of zero degrees along its entire length. One end of the rod is then suddenly subjected to a temperature of 100° and is maintained at that value. You may assume that the other end of the rod is perfectly insulated (i.e. there is no temperature gradient at that point). Using two 1D 'rod' elements, and assuming time-stepping parameters Δt

and θ, set up (but do not solve) the recursive matrix equations that will model the change in temperature along the rod as a function of time.
(Ans: $([\mathbf{M}_m] + \theta\Delta t\,[\mathbf{K}_c])\,\{\mathbf{\Phi}\}_1 = ([\mathbf{M}_m] - (1-\theta)\Delta t\,[\mathbf{K}_c])\,\{\mathbf{\Phi}\}_0$)

where

$$[\mathbf{M}_m] = \frac{1}{12}\begin{bmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{bmatrix} \text{ and } [\mathbf{K}_c] = \begin{bmatrix} 2 & -2 & 0 \\ -2 & 4 & -2 \\ 0 & -2 & 2 \end{bmatrix}$$

3. A layer of saturated soil is subjected at time $t = 0$ to a triangular excess pore pressure distribution varying from 60 units at the top to zero at the bottom. During the subsequent dissipation phase, the top and bottom of the layer can be considered to be fully drained. Using the 3-element discretisation shown in Figure 8.28 and a finite difference scheme with $\theta = 0.5$, estimate the pore pressures at the nodes after 0.1 secs using a single time step of $\Delta t = 0.1$. You may assume the excess pore pressure at the top of the stratum equals zero at all times (including $t = 0$).
(Ans: 35.6, 20.9 (Consistent); 38.6, 20.0 (Lumped); 37.0, 20.0 (Lumped-explicit))

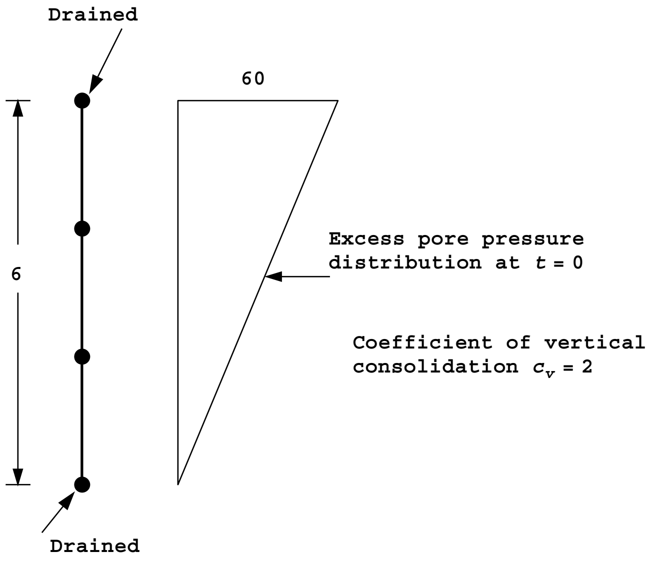

Drained
60
6
Excess pore pressure distribution at $t = 0$
Coefficient of vertical consolidation $c_v = 2$
Drained

Figure 8.28

4. A rod of length 1 unit and thermal diffusivity 1 unit is initially at a temperature of zero degrees along its entire length. One end of the rod is then subjected to a temperature, which increases linearly at a rate of 1°/second while the other end of the rod is maintained at 0°.

Using two 1D ‘rod’ elements as shown in Figure 8.29, and assuming a time step of $\Delta t = 1$ and a weighting factor of $\theta = 0.5$, use two steps to estimate the temperature at the central node after 2 s.
(Ans: $T(1) = 0.393,\ T(2) = 0.969$)

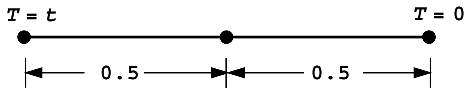


Figure 8.29

5. The rod shown in Figure 8.30 has an initially triangular temperature distribution varying linearly from zero degrees at each end to 100 degrees at the centre. The rod also has a variable diffusivity property as indicated. The rod is allowed to cool while maintaining the ends at zero degrees. Using a single finite element, and a "lumped mass" discretisation, estimate the temperature after 0.2 s at $x = 2$ and $x = 4$ along the rod.

   Use one time step of $\Delta t = 0.2$ with a time scaling parameter $\theta = 0.5$.
   (Ans: 2-element solution, 47.2, 88.6; 1-element solution, 46.4, 92.8)

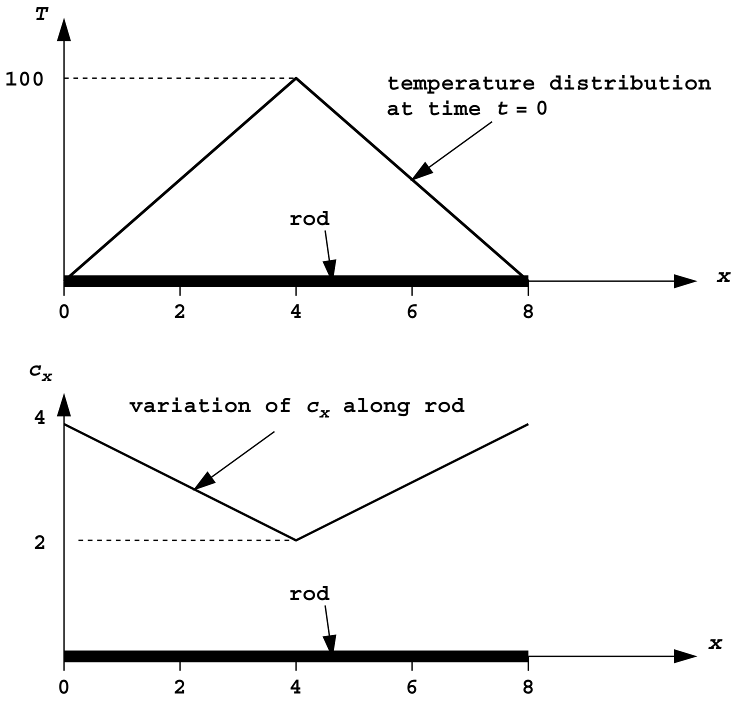


Figure 8.30

6. Starting with the governing 2D diffusion equation, go through the Galerkin weighted residual approach to derive terms k_{34} and m_{34} of the conductivity and "mass" matrices of the element shown in Figure 8.31.
   (Ans: $k_{34} = c_x b/(6a) - c_y a/(3b)$ $\quad m_{34} = ab/18$)

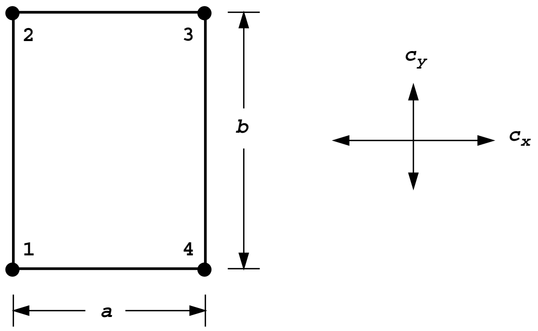


Figure 8.31

7. A rod of length D and thermal diffusivity c_x is initially at a uniform temperature of q_o when it is suddenly subjected to a boundary temperature of $q = 0$ at one end. Assuming the other end of the rod is perfectly insulated, use a single finite element and a lumped "mass" discretisation to estimate the dimensionless time it will take for the temperature at the insulated end of the rod to cool to one half of its initial value.
(Ans: $T = 0.35$)

8. A rod of unit length and unit thermal diffusivity is at an initial temperature, which varies linearly from zero degrees at one end to 100 degrees at the other. The cold end of the rod is perfectly insulated. At time $t = 0$ the hot end of the rod is suddenly changed to a temperature of zero degrees and maintained at that value. Use a 2-element discretisation and an explicit "lumped mass" algorithm to estimate the temperature at the insulated end of the rod after 0.01 and 0.02 s.
(Ans: $T(0.01) = 4.0$, $T(0.02) = 7.4$)

9. The rectangular plate shown in Figure 8.32 is initially at zero degrees when the boundaries are suddenly set to 50 degrees and maintained at that value. Select a simple 2D finite element discretisation, and hence estimate the time it takes for the temperature at the centre of the plate to rise to 25 degrees. Use an implicit scheme and consistent "mass" with $\theta = 0.5$ and $\Delta t = 0.05$ s.
(Ans: $t_{50\%} = 0.264$ secs.)

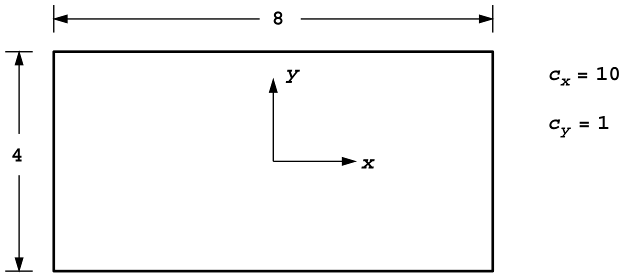


Figure 8.32

10. A rectangle of saturated soil has an x-dimension of 4 units and a y-dimension of 2 units. If drainage is allowed from all four faces of the rectangle and the initial excess pore pressure is set at all points to $u = 1.0$, use a single finite element and assume symmetry, to estimate the variation of u with time at the centre of the rectangle if $c_x = 10$ and $c_y = 1$.
    (Ans: Analytical, $u = e^{-21t/2}$)

11. A layer of saturated clay of Depth D and coefficient of consolidation c_v is drained at its top surface only. The layer is subjected to a sudden excess pore pressure which varies linearly from zero at the surface to U_o at the base. Using two 1D elements across the soil depth and a dimensionless time step of $\Delta T = 0.1$, estimate the average degree of consolidation when $T = 0.2$.
    (Ans: Using consistent "mass", $U = 0.4$)

References

Carslaw HS and Jaeger JC 1959 *Conduction of Heat in Solids*. Clarendon Press, Oxford.

Dobbins WE 1944 Effect of turbulence on sedimentation. *Trans Am Soc Civil Eng* **109**, 629–656.

Smith IM 1976 Integration in time of diffusion and diffusion-convection equations. *Finite Elements in Water Resources*, vol. 1. Pentech Press, Plymouth, pp. 3–20.

Smith IM 1979 The diffusion-convection equation. *Summary of Numerical Methods for Partial Differential Equations*. Oxford University Press. Chapter 11, pp. 195–211.

Smith IM, Farraday RV and O'Connor BA 1973 Rayleigh-Ritz and Galerkin finite elements for diffusion-convection problems. *Water Resour Res* **9**(3), 593–606.

9

Coupled Problems

9.1 Introduction

In the previous Chapter, flow problems were treated in terms of a single dependent variable, for example the "potential" or "total head", and solutions involved only 1 degree of freedom per node in the finite element mesh. While this simplification may be adequate in some cases, it may be necessary to solve problems in which several degrees of freedom exist at the nodes of the mesh and the several dependent variables, for example velocities and pressures, or displacements and pressures, are "coupled" in the differential equations. Strictly speaking, the equations of two- and three-dimensional elasticity involve coupling between the various components of displacement, but the term "coupled problems" is really reserved for those in which variables of entirely different types are interdependent.

Both steady state and transient problems are considered in this Chapter. As usual, the former involves the solution of sets of simultaneous equations, as in Chapters 4 to 7. Program 9.1 describes a steady state solution of the Navier–Stokes equations (see Sections 2.16, 3.11), in which the simultaneous equations are non-linear. An iterative process is therefore necessary during which the equations are solved repeatedly until the velocities and pressures have converged. As discussed in Section 3.11, these equations will have unsymmetrical coefficient matrices.

Program 9.2 solves the same problem without any global matrix assembly using a BiCGStab(l) iterative solver. In this case nested iterative processes are employed, with an internal one for BiCGStab iterations and an external one until convergence of velocities and pressures is obtained. The BiCGStab process is described in Section 3.5.3.

The remaining three programs describe coupled transient problems governed by the "Biot" equations (see Sections 2.18, 3.12). These coupled equations are cast as (linear) first order differential equations in the time variable, and solved by the implicit integration techniques introduced in Chapter 8.

Program 9.3 describes analysis of poro-elastic materials subjected to incremental time-dependent external loading (see Section 3.12.2). Program 9.4 enables investigations to be made of poro-elastic-plastic materials and transient collapse problems, by extending the previous program to include non-linear material behaviour governed by a Mohr–Coulomb failure criterion. Program 9.5 returns to poro-elasticity, and illustrates an absolute loading version of the Biot algorithm (see Section 3.12.1) using a "mesh free" approach involving a pcg solver.

Program 9.1 Analysis of the plane steady state Navier–Stokes equation using 8-node rectangular quadrilaterals for velocities coupled to 4-node rectangular quadrilaterals for pressures. Mesh numbered in *x*- or *y*-direction. Freedoms numbered in the order *u*-*p*-*v*.

```
PROGRAM p91
!-------------------------------------------------------------------------
! Program 9.1 Analysis of the plane steady state Navier-Stokes equation
!             using 8-node rectangular quadrilaterals for velocities
!             coupled to 4-node rectangular quadrilaterals for pressures.
!             Mesh numbered in x- or y-direction. Freedoms numbered in the
!             order u-p-v.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::fixed_freedoms,i,iel,iters,k,limit,nband,ndim=2,nels,neq,nip=4, &
   nn,nod=8,nodf=4,nodof=3,nr,ntot=20,nxe,nye
 REAL(iwp)::det,one=1.0_iwp,penalty=1.e20_iwp,pt5=0.5_iwp,rho,tol,ubar,   &
  vbar,visc,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::g(:),g_g(:,:),g_num(:,:),nf(:,:),no(:),node(:),     &
   num(:),sense(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),coordf(:,:),c11(:,:),c12(:,:),c21(:,:),&
   c23(:,:),c32(:,:),der(:,:),derf(:,:),deriv(:,:),derivf(:,:),fun(:),    &
   funf(:),g_coord(:,:),jac(:,:),kay(:,:),ke(:,:),loads(:),nd1(:,:),      &
   nd2(:,:),ndf1(:,:),ndf2(:,:),nfd1(:,:),nfd2(:,:),oldlds(:),pb(:,:),    &
   points(:,:),uvel(:),value(:),vvel(:),weights(:),work(:,:),x_coords(:), &
   y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,tol,limit,visc,rho
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),uvel(nod),   &
   jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),vvel(nod), &
   derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),g_g(ntot,nels),           &
   c11(nod,nod),c12(nod,nodf),c21(nodf,nod),c23(nodf,nod),g(ntot),        &
   c32(nod,nodf),ke(ntot,ntot),fun(nod),x_coords(nxe+1),y_coords(nye+1),  &
   nf(nodof,nn),g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip),   &
   nd1(nod,nod),nd2(nod,nod),ndf1(nod,nodf),ndf2(nod,nodf),nfd1(nodf,nod),&
   nfd2(nodf,nod))
 READ(10,*)x_coords,y_coords
 uvel=zero; vvel=zero; kay=zero; kay(1,1)=visc/rho; kay(2,2)=visc/rho
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 CALL sample(element,points,weights); nband=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   g(1:8)=nf(1,num(1:8)); g(9:12)=nf(2,num(1:7:2)); g(13:20)=nf(3,num(1:8))
   g_num(:,iel)=num;g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
   IF(nband<bandwidth(g))nband=bandwidth(g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the full bandwidth is",2*(nband+1)-1
 ALLOCATE(pb(neq,2*(nband+1)-1),loads(0:neq),oldlds(0:neq),               &
   work(nband+1,neq)); loads=zero; oldlds=zero; iters=0
 READ(10,*)fixed_freedoms
 ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                     &
   value(fixed_freedoms), no(fixed_freedoms))
 READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
!-----------------------iteration loop----------------------------------
 iterations: DO
   iters=iters+1; converged=.FALSE.; pb=zero; work=zero; ke=zero
!-----------------------global matrix assembly----------------------------
   elements_2: DO iel=1,nels
     num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
     coordf=coord(1:7:2,:); uvel=(loads(g(1:nod))+oldlds(g(1:nod)))*pt5
     DO i=nod+nodf+1,ntot
       vvel(i-nod-nodf)=(loads(g(i))+oldlds(g(i)))*pt5
     END DO; c11=zero; c12=zero; c21=zero; c23=zero; c32=zero
     gauss_points_1: DO i=1,nip
!-----------------------velocity contribution-----------------------------
       CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
       jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
       deriv=MATMUL(jac,der)
       ubar=DOT_PRODUCT(fun,uvel); vbar=DOT_PRODUCT(fun,vvel)
       IF(iters==1)THEN; ubar=one; vbar=zero; END IF
       CALL cross_product(fun,deriv(1,:),nd1)
       CALL cross_product(fun,deriv(2,:),nd2)
       c11=c11+det*weights(i)*(MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)+&
         nd1*ubar+nd2*vbar)
!-----------------------pressure contribution-----------------------------
       CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
       jac=MATMUL(derf,coordf); det=determinant(jac); CALL invert(jac)
       derivf=MATMUL(jac,derf)
       CALL cross_product(fun,derivf(1,:),ndf1)
       CALL cross_product(fun,derivf(2,:),ndf2)
       CALL cross_product(funf,deriv(1,:),nfd1)
       CALL cross_product(funf,deriv(2,:),nfd2)
       c12=c12+ndf1*det*weights(i)/rho; c32=c32+ndf2*det*weights(i)/rho
       c21=c21+nfd1*det*weights(i); c23=c23+nfd2*det*weights(i)
     END DO gauss_points_1
     CALL formupv(ke,c11,c12,c21,c23,c32); CALL formtb(pb,ke,g)
   END DO elements_2; loads=zero
!-----------------------specify pressure and velocity boundary values-----
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   pb(no,nband+1)=pb(no,nband+1)+penalty; loads(no)=pb(no,nband+1)*value
!-----------------------equation solution---------------------------------
   CALL gauss_band(pb,work); CALL solve_band(pb,work,loads); loads(0)=zero
   CALL checon(loads,oldlds,tol,converged)
   IF(converged.OR.iters==limit)EXIT
 END DO iterations
 WRITE(11,'(/A)')"  Node     u-velocity  pressure    v-velocity"
 DO k=1,nn; WRITE(11,'(I5,A,3E12.4)')k,"    ",loads(nf(:,k)); END DO
 WRITE(11,'(/A,I3,A)')" Converged in",iters," iterations."
 nf(2,:)=nf(3,:); CALL vecmsh(loads,nf,0.3_iwp,0.05_iwp,g_coord,g_num,14)
STOP
END PROGRAM p91
```

Scalar integers:

`fixed_freedoms`	number of fixed freedoms
`i`	simple counter
`iel`	simple counter
`iters`	iteration counter
`iwp`	`SELECTED_REAL_KIND(15)`
`k`	simple counter
`limit`	iteration ceiling
`nband`	full bandwidth of non-symmetric matrix
`ndim`	number of dimensions
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integrating points
`nn`	number of nodes in the mesh
`nod`	number of nodes per solid element
`nodf`	number of nodes per fluid element
`nodof`	number of degrees of freedom per node
`nr`	number of restrained nodes
`ntot`	total number of degrees of freedom per element
`nxe`	number of elements in x-direction
`nye`	number of elements in y-direction

Scalar reals:

`det`	determinant of the Jacobian matrix
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`pt5`	set to 0.5
`rho`	fluid density
`tol`	convergence tolerance
`ubar`	average x-velocity
`vbar`	average y-velocity
`visc`	molecular viscosity
`zero`	set to 0.0

Character variables:

`element`	element type

Scalar logical:

`converged`	set to `.TRUE.` if solution converged

Dynamic integer arrays:

`g`	element "steering" vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`nf`	nodal freedoms
`no`	freedoms to be fixed

node	nodes with fixed values
num	element node numbers
sense	sense of freedom to be fixed

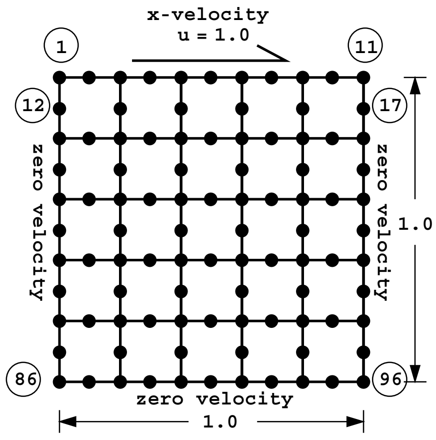
Dynamic real arrays:

coord	solid element nodal coordinates
coordf	fluid element nodal coordinates
c11	element submatrix (2.115)
c12	element submatrix (2.115)
c21	element submatrix (2.115)
c23	element submatrix (2.115)
c32	element submatrix (2.115)
der	solid shape function derivatives wrt local coordinates
derf	fluid shape function derivatives wrt local coordinates
deriv	solid shape function derivatives wrt global coordinates
derivf	fluid shape function derivatives wrt global coordinates
fun	solid shape functions
funf	fluid shape functions
g_coord	nodal coordinates for all elements
jac	Jacobian matrix
kay	property matrix
ke	element "stiffness" matrix
loads	nodal velocities and pressures
nd1	product [fun]^T[deriv(1, :)]
nd2	product [fun]^T[deriv(2, :)]
ndf1	product [fun]^T[derivf(1, :)]
ndf2	product [fun]^T[derivf(2, :)]
nfd1	product [funf]^T[deriv(1, :)]
nfd2	product [funf]^T[deriv(2, :)]
oldlds	nodal velocities and pressures from previous iteration
pb	unsymmetric global band "stiffness" matrix
points	integrating point local coordinates
uvel	element nodal x-velocity
value	fixed vales of freedoms
vvel	element nodal y-velocity
weights	weighting coefficients
work	working space
x_coords	x-coordinates of mesh layout
y_coords	y-coordinates of mesh layout

The (steady state) Navier–Stokes equations were developed in Chapter 2, (Section 2.16). The equilibrium equations to be solved are (2.114), whose coefficients are themselves functions of the velocities u and v so that the equations are non-linear. Furthermore, the coefficient submatrices $[\mathbf{c}_{ij}]$ are not, in general, symmetrical and reference to Section 3.11 shows that the subroutines gauss_band and solve_band will be required to operate on the banded equation coefficients.

Section 3.11 illustrates how the element submatrices $[\mathbf{c}_{ij}]$ are assembled and uses much of the program terminology already developed for uncoupled flow problems in Chapters 7 and 8.

The simple problem chosen to illustrate this program is shown in Figure 9.1. Flow is confined to a rectangular cavity and driven by a uniform horizontal velocity at the top. The velocities at the other three boundaries are set to zero. Eight node elements are used to model the vector field of velocities, and 4-node elements are used to model the scalar field of pressures. Note that a dummy freedom has been inserted at all mid-side nodes where



```
nxe   nye   tol   limit   visc   rho
5      5  0.001   30     0.01   1.0

x_coords, y_coords
0.0    0.2    0.4    0.6    0.8    1.0
0.0   -0.2   -0.4   -0.6   -0.8   -1.0

nr,(k,nf(:,k),i=1,nr)
81
 1 1 0 0    2 1 0 0    3 1 1 0    4 1 0 0    5 1 1 0    6 1 0 0
 7 1 1 0    8 1 0 0    9 1 1 0   10 1 0 0   11 1 1 0   12 0 0 0
13 1 0 1   14 1 0 1   15 1 0 1   16 1 0 1   17 0 0 0   18 0 1 0
19 1 0 1   21 1 0 1   23 1 0 1   25 1 0 1   27 1 0 1   28 0 1 0
29 0 0 0   30 1 0 1   31 1 0 1   32 1 0 1   33 1 0 1   34 0 0 0
35 0 1 0   36 1 0 1   38 1 0 1   40 1 0 1   42 1 0 1   44 1 0 1
45 0 1 0   46 0 0 0   47 1 0 1   48 1 0 1   49 1 0 1   50 1 0 1
51 0 0 0   52 0 1 0   53 1 0 1   54 1 1 1   55 1 0 1   57 1 0 1
59 1 0 1   61 1 0 1   62 0 1 0   63 0 0 0   64 1 0 1   65 1 0 1
66 1 0 1   67 1 0 1   68 0 0 0   69 0 1 0   70 1 0 1   72 1 0 1
74 1 0 1   76 1 0 1   78 1 0 1   79 0 1 0   80 0 0 0   81 1 0 1
82 1 0 1   83 1 0 1   84 1 0 1   85 0 0 0   86 0 1 0   87 0 0 0
88 0 1 0   89 0 0 0   90 0 1 0   91 0 0 0   92 0 1 0   93 0 0 0
94 0 1 0   95 0 0 0   96 0 1 0

fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
11
 1  1  1.0     2  1  1.0     3  1  1.0     4  1  1.0     5  1  1.0
 6  1  1.0     7  1  1.0     8  1  1.0     9  1  1.0    10  1  1.0
11  1  1.0
```

Figure 9.1 Mesh and data for Program 9.1 example

there is no p variable. Thus, the second freedom (nodal freedoms are in the order u, p, v) at all mid-side nodes is eliminated from the analysis through the “restrained freedom” (`nf`) data.

The first line of data reads the number of elements in each direction of the rectangular mesh (`nxe` and `nye`), the tolerance (`tol`) and iteration ceiling (`limit`) for the non-linear iterations, and the fluid properties given by the molecular viscosity (`visc`) and density (`rho`). The second and third lines give the coordinate data `x_coords` and `y_coords`, and this is followed by the node freedom data, which involves `nr=81` restrained nodes in this example. The final lines of data give the fixed velocity boundary condition of $u = 1.0$ along the 11 top nodes of the mesh.

The structure of the program is described by the structure chart in Figure 9.2. After the data has been read in, various arrays must be initialised to zero. Note that the `kay` matrix which, in the previous chapter, held coefficients of consolidation c_x and so on, now holds viscosity and density in the form μ/ρ, which represents the reciprocal of the Reynolds number for this cavity size.

The iteration loop is then entered, controlled by the counter `iters`. System arrays `pb` and `work` must be nulled together with the element “stiffness” matrix `ke`. Element matrix

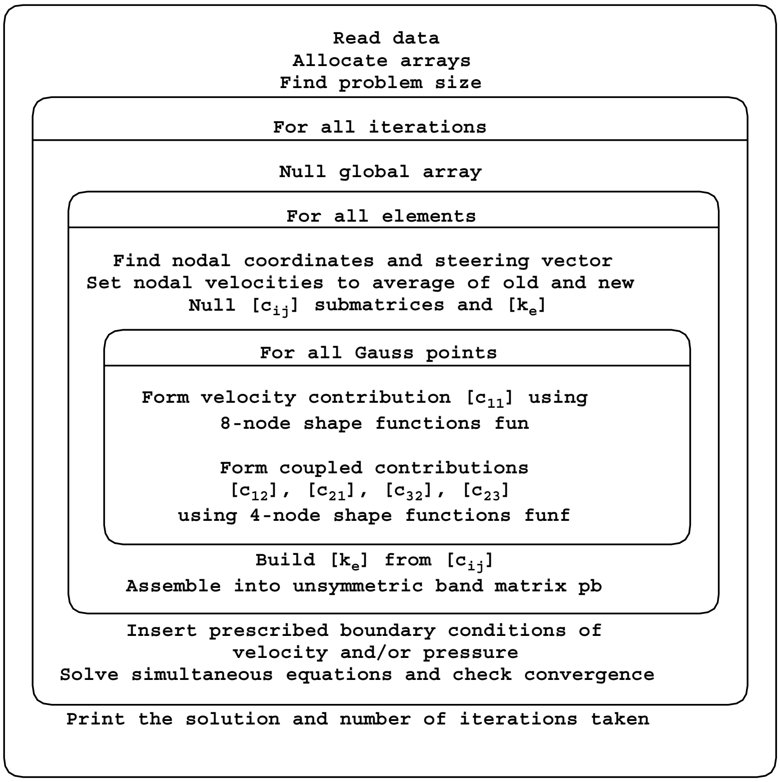

Read data
Allocate arrays
Find problem size
For all iterations
Null global array
For all elements
Find nodal coordinates and steering vector
Set nodal velocities to average of old and new
Null [c_{ij}] submatrices and [k_e]
For all Gauss points
Form velocity contribution [c_{11}] using
8-node shape functions fun
Form coupled contributions
[c_{12}], [c_{21}], [c_{32}], [c_{23}]
using 4-node shape functions funf
Build [k_e] from [c_{ij}]
Assemble into unsymmetric band matrix pb
Insert prescribed boundary conditions of
velocity and/or pressure
Solve simultaneous equations and check convergence
Print the solution and number of iterations taken

Figure 9.2 Structure chart for Navier–Stokes analysis with global matrix assembly in Program 9.1

integration and assembly then proceeds as usual. The nodal coordinates and steering vector are formed as usual by the geometry library subroutine `geom_rect` with numbering in this case in the x-direction. Nodal velocities used to form $\overline{u}$ and $\overline{v}$ in equation (2.115) are taken to be the average of those computed in the last two iterations. Element submatrices `c11`, and so on, are set to zero and the numerical integration loop entered. Average velocities $\overline{u}$ and $\overline{v}$ are recovered form `uvel` and `vvel`, except in the first iteration where the "guess" $\overline{u} = 1.0$ and $\overline{v} = 0.0$ is used.

The submatrix `c11` is formed as required by equations (3.103–3.104), followed by submatrices `c12`, `c32`, `c21`, and `c23` as demanded by equation (3.105–3.106). The element "stiffness" `ke` is built from the submatrices (2.114) by the subroutine `formupv` and assembled into the global unsymmetrical band matrix `pb` by the assembly subroutine `formtb`.

It remains only to specify the fixed freedoms by the "penalty" technique (see Section 3.6) and to complete the equation solution using subroutines `gauss_band` and `solve_band`. The maximum number of iterations allowed is 30 (`limit`) but a convergence check of 0.001 (`tol`) is invoked by subroutine `checon`.

The results are listed as Figure 9.3 and velocity vectors generated by subroutine `vecmsh` and output to file `fe95.vec` illustrated in Figure 9.4. Regridding strategies (Kidger, 1994) can further enhance the visualisation. As the Reynolds number increases, convergence of this algorithm will be slower.

```
There are  158 equations and the full bandwidth is   79

 Node     u-velocity  pressure    v-velocity
   1      0.1000E+01  0.0000E+00  0.0000E+00
   2      0.1000E+01  0.0000E+00  0.0000E+00
   3      0.1000E+01  0.2429E+00  0.0000E+00
   4      0.1000E+01  0.0000E+00  0.0000E+00
   5      0.1000E+01  0.1704E+00  0.0000E+00
   6      0.1000E+01  0.0000E+00  0.0000E+00
   7      0.1000E+01  0.2096E+00  0.0000E+00
   8      0.1000E+01  0.0000E+00  0.0000E+00
   9      0.1000E+01  0.2101E+00  0.0000E+00
  10      0.1000E+01  0.0000E+00  0.0000E+00
  11      0.1000E+01  0.8023E+00  0.0000E+00
  12      0.0000E+00  0.0000E+00  0.0000E+00
  13      0.2073E+00  0.0000E+00  0.5662E-01
  14      0.3854E+00  0.0000E+00  0.5448E-01
  15      0.4901E+00  0.0000E+00 -0.7326E-02
.
.
.
  90      0.0000E+00  0.2199E+00  0.0000E+00
  91      0.0000E+00  0.0000E+00  0.0000E+00
  92      0.0000E+00  0.2236E+00  0.0000E+00
  93      0.0000E+00  0.0000E+00  0.0000E+00
  94      0.0000E+00  0.2275E+00  0.0000E+00
  95      0.0000E+00  0.0000E+00  0.0000E+00
  96      0.0000E+00  0.2201E+00  0.0000E+00

Converged in  7 iterations.
```

Figure 9.3 Results from Program 9.1 example

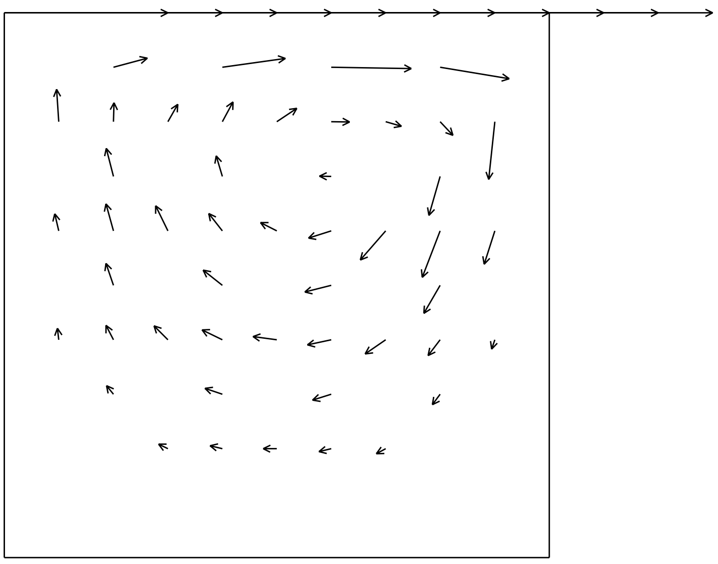


Figure 9.4 Velocity vectors at convergence from Program 9.1 example

Program 9.2 Analysis of the plane steady state Navier–Stokes equation using 8-node rectangular quadrilaterals for velocities coupled to 4-node rectangular quadrilaterals for pressures. Mesh numbered in x- or y-direction. Freedoms numbered in the order u-p-v. Element-by-element solution using BiCGStab(l) with no preconditioning. No global matrix assembly.

```
PROGRAM p92
!-------------------------------------------------------------------------
! Program 9.2 Analysis of the plane steady state Navier-Stokes equation
!             using 8-node rectangular quadrilaterals for velocities
!             coupled to 4-node rectangular quadrilaterals for pressures.
!             Mesh numbered in x- or y-direction. Freedoms numbered in the
!             order u-p-v. Element by element solution using BiCGStab(l).
!             with no preconditioning. No global matrix assembly,
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,cg_tot,ell,fixed_freedoms,i,iel,iters,j,k,    &
   limit,ndim=2,nels,neq,nip=4,nn,nod=8,nodf=4,nodof=3,nr,ntot=20,nxe,nye
 REAL(iwp)::alpha,beta,cg_tol,det,error,gama,kappa,norm_r,omega,          &
   one=1.0_iwp,penalty=1.e5_iwp,pt5=0.5_iwp,rho,rho1,r0_norm,tol,ubar,    &
   vbar,visc,x0,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
 LOGICAL::converged,cg_converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::g(:),g_g(:,:),g_num(:,:),nf(:,:),no(:),node(:),     &
   num(:),sense(:)
 REAL(iwp),ALLOCATABLE::b(:),coord(:,:),coordf(:,:),c11(:,:),c12(:,:),    &
   c21(:,:),c23(:,:),c32(:,:),der(:,:),derf(:,:),deriv(:,:),derivf(:,:),  &
   diag(:),fun(:),funf(:),gamma(:),gg(:,:),g_coord(:,:),hh(:,:),jac(:,:), &
   kay(:,:),ke(:,:),loads(:),nd1(:,:),nd2(:,:),nfd1(:,:),nfd2(:,:),       &
   ndf1(:,:),ndf2(:,:),oldlds(:),points(:,:),r(:,:),rt(:),s(:),store(:),  &
   storke(:,:,:),u(:,:),uvel(:),value(:),vvel(:),weights(:),x_coords(:),  &
   y(:),y1(:),y_coords(:)
!-----------------------input and initialisation------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,tol,limit,visc,rho,cg_tol,cg_limit,x0,ell,kappa
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),             &
   jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),           &
   derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),nd1(nod,nod),nd2(nod,nod),&
   ndf1(nod,nodf),ndf2(nod,nodf),nfd1(nodf,nod),nfd2(nodf,nod),           &
   g_g(ntot,nels),c11(nod,nod),c12(nod,nodf),c21(nodf,nod),g(ntot),       &
   ke(ntot,ntot),fun(nod),x_coords(nxe+1),y_coords(nye+1),nf(nodof,nn),   &
   g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip),c32(nod,nodf),  &
   c23(nodf,nod),uvel(nod),vvel(nod),storke(ntot,ntot,nels),s(ell+1),     &
   gg(ell+1,ell+1),gamma(ell+1))
 READ(10,*)x_coords,y_coords
 uvel=zero; vvel=zero; kay=zero; kay(1,1)=visc/rho; kay(2,2)=visc/rho
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 CALL sample(element,points,weights)
!-----------------------loop the elements to set up global arrays---------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   g(:8)=nf(1,num(:8)); g(9:12)=nf(2,num(1:7:2)); g(13:20)=nf(3,num(:8))
   g_num(:,iel )=num; g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 WRITE(11,'(A,I5,A)')" There are",neq," equations"
 ALLOCATE(loads(0:neq),rt(0:neq),r(0:neq,ell+1),u(0:neq,ell+1),b(0:neq),  &
   diag(0:neq),oldlds(0:neq),y(0:neq),y1(0:neq))
 READ(10,*)fixed_freedoms
 ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                     &
   value(fixed_freedoms),no(fixed_freedoms),store(fixed_freedoms))
 READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
 iters=0; cg_tot=0; loads=zero; oldlds=zero
!-----------------------iteration loop-----------------------------------
 iterations: do
   iters=iters+1; converged=.FALSE.; ke=zero; diag=zero; b=zero
!-----------------------element stiffness integration and storage --------
   elements_2: DO iel=1,nels
     num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
     coordf=coord(1:7:2,:); uvel=(loads(g(:nod))+oldlds(g(:nod)))*pt5
     DO i=nod+nodf+1,ntot
       vvel(i-nod-nodf)=(loads(g(i))+oldlds(g(i)))*pt5
     END DO; c11=zero; c12=zero; c21=zero; c23=zero; c32=zero
     gauss_points_1: DO i=1,nip
!-----------------------velocity contribution----------------------------
       CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
       jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
       deriv=MATMUL(jac,der)
       ubar=DOT_PRODUCT(fun,uvel); vbar=DOT_PRODUCT(fun,vvel)
       IF(iters==1)THEN; ubar=one; vbar=zero; END IF
       CALL cross_product(fun,deriv(1,:),nd1)
       CALL cross_product(fun,deriv(2,:),nd2)
       c11=c11+det*weights(i)*(MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)+ &
         nd1*ubar+nd2*vbar)
!-----------------------pressure contribution----------------------------
       CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
       jac=MATMUL(derf,coordf); det=determinant(jac); CALL invert(jac)
       derivf=MATMUL(jac,derf)
       CALL cross_product(fun,derivf(1,:),ndf1)
       CALL cross_product(fun,derivf(2,:),ndf2)
       CALL cross_product(funf,deriv(1,:),nfd1)
       CALL cross_product(funf,deriv(2,:),nfd2)
       c12=c12+ndf1*det*weights(i)/rho; c32=c32+ndf2*det*weights(i)/rho
       c21=c21+nfd1*det*weights(i); c23=c23+nfd2*det*weights(i)
     END DO gauss_points_1
     CALL formupv(ke,c11,c12,c21,c23,c32); storke(:,:,iel)=ke
     DO k=1,ntot; diag(g(k))=diag(g(k))+ke(k,k); END DO
   END DO elements_2
!------------------------specify pressure and velocity boundary values-----
   DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
   diag(no)=diag(no)+penalty; b(no)=diag(no)*value; store=diag(no)
!---solve the simultaneous equations element by element using BiCGStab(l)-
!------------------------initialisation phase------------------------------
   IF(iters==1)loads=x0; loads(0)=zero; y=loads; y1=zero
   elements_3: DO iel=1,nels
     g=g_g(:,iel); ke=storke(:,:,iel); y1(g)=y1(g)+MATMUL(ke,y(g))
   END DO elements_3; cg_iters=0
   y1(0)=zero; y1(no)=y(no)*store; y=y1; rt=b-y; r=zero; r(:,1)=rt; u=zero
   gama=one; omega=one; k=0; norm_r=norm(rt); r0_norm=norm_r; error=one
!------------------------BiCGStab(l) iterations----------------------------
   bicg_iterations: DO
     cg_iters=cg_iters+1; cg_converged=error<cg_tol
     IF(cg_iters==cg_limit.OR.cg_converged)EXIT
     gama=-omega*gama; y=r(:,1)
     DO j=1,ell
       rho1=DOT_PRODUCT(rt,y); beta=rho1/gama
       u(:,1:j)=r(:,1:j)-beta*u(:,1:j); y=u(:,j); y1=zero
       elements_4: DO iel=1,nels
         g=g_g(:,iel); ke=storke(:,:,iel); y1(g)=y1(g)+MATMUL(ke,y(g))
       END DO elements_4
       y1(0)=zero; y1(no)=y(no)*store; y=y1; u(:,j+1)=y
       gama=DOT_PRODUCT(rt,y); alpha=rho1/gama; loads=loads+alpha*u(:,1)
       r(:,1:j)=r(:,1:j)-alpha*u(:,2:j+1); y=r(:,j); y1=zero
       elements_5: DO iel=1,nels
         g=g_g(:,iel); ke=storke(:,:,iel); y1(g)=y1(g)+MATMUL(ke,y(g))
       END DO elements_5; y1(0)=zero; y1(no)=y(no)*store; y=y1; r(:,j+1)=y
     END DO; gg=MATMUL(TRANSPOSE(r),r)
     CALL form_s(gg,ell,kappa,omega,gamma,s); loads=loads-MATMUL(r,s)
     r(:,1)=MATMUL(r,gamma); u(:,1)=MATMUL(u,gamma); norm_r=norm(r(:,1))
     error=norm_r/r0_norm; k=k+1
   END DO bicg_iterations; cg_tot=cg_tot+cg_iters
!------------------------end of BiCGStab(l) process-----------------------
   CALL checon(loads,oldlds,tol,converged)
   IF(converged.OR.iters==limit)EXIT
 END DO iterations
 WRITE(11,'(/A)')"  Node     u-velocity  pressure    v-velocity"
 DO k=1,nn; WRITE(11,'(I5,A,3E12.4)')k,"    ",loads(nf(:,k)); END DO
 WRITE(11,'(/A,I3,A/A,F6.2,A)')" Converged in",iters," iterations",       &
   " with an average of", REAL(cg_tot/iters), "  BiCGStab(l) iterations."
 nf(2,:)=nf(3,:); CALL vecmsh(loads,nf,0.3_iwp,0.05_iwp,g_coord,g_num,14)
STOP
END PROGRAM p92
```

New scalar integers:

`cg_iters`	BiCGStab iteration counter
`cg_limit`	BiCGStab iteration ceiling
`cg_tot`	holds total number of BiCGStab iterations
`ell`	BiCGStab parameter; taken as 4 in this example
`j`	simple counter
`k`	simple counter

New scalar reals:

`alpha`	local variable
`beta`	local variable
`cg_tol`	BiCGStab convergence tolerance
`error`	measure of error in BiCGStab
`gama`	local variable
`kappa`	BiCGStab parameter; taken as zero in this example
`norm_r`	residual norm
`omega`	local variable
`rho1`	local variable
`r0_norm`	initial residual norm
`x0`	initialisation value

New scalar logical:

`cg_converged`	set to `.TRUE` if BiCGStab has converged

New dynamic real arrays:

`b`	right-hand side vector
`diag`	diagonal of left-hand side matrix
`gamma`	small local array
`gg`	small local array
`r`	residual vector
`rt`	initial residual vector
`s`	small local vector
`store`	"penalty" degrees of freedom
`storke`	element matrix storage
`u`	gather/scatter array
`y`	gather/scatter array
`y1`	gather/scatter array

The non-symmetric structures of the element and global matrices generated by Program 9.1 require a different iterative algorithm to the pcg methods used for symmetric equations earlier in the text. In Program 9.2 the BiCGStab(l) algorithm (see Section 3.5.3) is used to re-solve the same cavity flow problem analysed by the previous program. The iterative algorithm requires no global matrix assembly and achieves all global matrix–vector products via gather/scatter algorithms at the element level. Compared with Figure 9.1 the data shown in Figure 9.5 includes an additional line which reads `cg_tol`, `cg_limit`, `x0`, `ell` and `kappa` described in the notation section above. The results shown in Figure 9.6 are essentially identical to those in Figure 9.3 using Program 9.1. The output in this case

```
nxe  nye  tol  limit  visc  rho
5     5  0.001  30    0.01  1.0

cg_tol  cg_limit  x0  ell  kappa
1.0e-5    200    1.0   4    0.0

x_coords, y_coords
0.0   0.2   0.4   0.6   0.8   1.0
0.0  -0.2  -0.4  -0.6  -0.8  -1.0

nr,(k,nf(:,k),i=1,nr)
81
 1 1 0 0   2 1 0 0   3 1 1 0   4 1 0 0   5 1 1 0   6 1 0 0
 7 1 1 0   8 1 0 0   9 1 1 0  10 1 0 0  11 1 1 0  12 0 0 0
13 1 0 1  14 1 0 1  15 1 0 1  16 1 0 1  17 0 0 0  18 0 1 0
19 1 0 1  21 1 0 1  23 1 0 1  25 1 0 1  27 1 0 1  28 0 1 0
29 0 0 0  30 1 0 1  31 1 0 1  32 1 0 1  33 1 0 1  34 0 0 0
35 0 1 0  36 1 0 1  38 1 0 1  40 1 0 1  42 1 0 1  44 1 0 1
45 0 1 0  46 0 0 0  47 1 0 1  48 1 0 1  49 1 0 1  50 1 0 1
51 0 0 0  52 0 1 0  53 1 0 1  54 1 1 1  55 1 0 1  57 1 0 1
59 1 0 1  61 1 0 1  62 0 1 0  63 0 0 0  64 1 0 1  65 1 0 1
66 1 0 1  67 1 0 1  68 0 0 0  69 0 1 0  70 1 0 1  72 1 0 1
74 1 0 1  76 1 0 1  78 1 0 1  79 0 1 0  80 0 0 0  81 1 0 1
82 1 0 1  83 1 0 1  84 1 0 1  85 0 0 0  86 0 1 0  87 0 0 0
88 0 1 0  89 0 0 0  90 0 1 0  91 0 0 0  92 0 1 0  93 0 0 0
94 0 1 0  95 0 0 0  96 0 1 0

fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
11
 1  1  1.0    2  1  1.0    3  1  1.0    4  1  1.0    5  1  1.0
 6  1  1.0    7  1  1.0    8  1  1.0    9  1  1.0   10  1  1.0
11  1  1.0
```

Figure 9.5 Data for Program 9.2 example

```
 There are  158 equations

  Node     u-velocity  pressure    v-velocity
    1      0.1000E+01  0.0000E+00  0.0000E+00
    2      0.1000E+01  0.0000E+00  0.0000E+00
    3      0.1000E+01  0.2429E+00  0.0000E+00
    4      0.1000E+01  0.0000E+00  0.0000E+00
    5      0.1000E+01  0.1704E+00  0.0000E+00
    6      0.1000E+01  0.0000E+00  0.0000E+00
    7      0.1000E+01  0.2096E+00  0.0000E+00
    8      0.1000E+01  0.0000E+00  0.0000E+00
    9      0.1000E+01  0.2101E+00  0.0000E+00
   10      0.1000E+01  0.0000E+00  0.0000E+00
   11      0.1000E+01  0.8023E+00  0.0000E+00
   12      0.0000E+00  0.0000E+00  0.0000E+00
   13      0.2073E+00  0.0000E+00  0.5662E-01
   14      0.3854E+00  0.0000E+00  0.5448E-01
   15      0.4901E+00  0.0000E+00 -0.7326E-02
.
.
.
   90      0.0000E+00  0.2199E+00  0.0000E+00
   91      0.0000E+00  0.0000E+00  0.0000E+00
   92      0.0000E+00  0.2236E+00  0.0000E+00
   93      0.0000E+00  0.0000E+00  0.0000E+00
   94      0.0000E+00  0.2275E+00  0.0000E+00
   95      0.0000E+00  0.0000E+00  0.0000E+00
   96      0.0000E+00  0.2201E+00  0.0000E+00

 Converged in  7 iterations
 with an average of 32.00  BiCGStab(l) iterations.
```

Figure 9.6 Results from Program 9.2 example

indicates that with `ell=4`, an average of 32 internal BiCGStab iterations were require for each of the seven external Navier–Stokes iterations.

Program 9.3 Plane strain consolidation analysis of a Biot poro-elastic solid using 8-node rectangular quadrilaterals for displacements coupled to 4-node rectangular quadrilaterals for pressures. Freedoms numbered in the order u-v-u_w. Incremental version.

```
PROGRAM p93
!-------------------------------------------------------------------------
! Program 9.3 Plane strain consolidation analysis of a Biot elastic
!             solid using 8-node rectangular quadrilaterals for
!             displacements coupled to 4-node rectangular quadrilaterals
!             for pressures. Incremental load version.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,j,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nlfp,nls,  &
   nn,nod=8,nodf=4,nodof=3,npri,nprops=4,np_types,nr,nres,nst=3,nstep,    &
   ntot=20,nxe,nye
 REAL(iwp)::det,dtim,theta,time,tot_load,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays-------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   no(:),num(:)
 REAL(iwp),ALLOCATABLE::al(:),ans(:),bee(:,:),c(:,:),coord(:,:),dee(:,:), &
   der(:,:),derf(:,:),deriv(:,:),derivf(:,:),eld(:),fun(:),funf(:),gc(:), &
   g_coord(:,:),jac(:,:),kay(:,:),ke(:,:),km(:,:),kc(:,:),kv(:),lf(:,:),  &
   loads(:),phi0(:),phi1(:),points(:,:),prop(:,:),sigma(:),               &
   store_kc(:,:,:),val(:,:),vol(:),volf(:,:),weights(:),x_coords(:),      &
   y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(dee(nst,nst),points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),&
   jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),           &
   derf(ndim,nodf),funf(nodf),bee(nst,ndof),km(ndof,ndof),kc(nodf,nodf),  &
   g_g(ntot,nels),ke(ntot,ntot),c(ndof,nodf),x_coords(nxe+1),             &
   y_coords(nye+1),vol(ndof),nf(nodof,nn),g(ntot),volf(ndof,nodf),        &
   g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip),                &
   store_kc(nodf,nodf,nels),phi0(nodf),phi1(nodf),prop(nprops,np_types),  &
   etype(nels),eld(ndof),gc(ndim),sigma(nst),fun(nod))
 READ(10,*)prop; etype=1; if(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),ans(0:neq))
 READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(no(i),val(i,:),i=1,loaded_nodes)
 READ(10,*)nlfp; ALLOCATE(lf(2,nlfp))
 READ(10,*)lf; nls=FLOOR(lf(1,nlfp)/dtim); IF(nstep>nls)nstep=nls
 ALLOCATE(al(nstep)); CALL load_function(lf,dtim,al); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   g(1:15:2)=nf(1,num(:)); g(2:16:2)=nf(2,num(:)); g(17:)=nf(3,num(1:7:2))
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                      &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 CALL sample(element,points,weights); loads=zero; kv=zero
!-----------------------global matrix assembly----------------------------
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; c=zero; kc=zero
 gauss_points_1: DO i=1,nip
!-----------------------elastic solid contribution------------------------
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     vol(:)=bee(1,:)+bee(2,:)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----------------------fluid contribution--------------------------------
     CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
     derivf=MATMUL(jac,derf)
     kc=kc+MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtim
     CALL cross_product(vol,funf,volf); c=c+volf*det*weights(i)
   END DO gauss_points_1; store_kc(:,:,iel)=kc
   CALL formke(km,kc,c,ke,theta); CALL fsparv(kv,ke,g,kdiag)
 END DO elements_2
!-----------------------factorise equations-------------------------------
 CALL sparin_gauss(kv,kdiag)
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/A,I5)')" Results at node",nres
 WRITE(11,'(A)')                                                            &
   "    time        load        x-disp      y-disp   porepressure"
 WRITE(11,'(5E12.4)')0.0,0.0,loads(nf(:,nres))
 time_steps: DO j=1,nstep
   tot_load=SUM(al(1:j)); time=j*dtim; ans=zero
   elements_3: DO iel=1,nels
     g=g_g(:,iel); kc=store_kc(:,:,iel)
     phi0=loads(g(ndof+1:)); phi1=MATMUL(kc,phi0)
     ans(g(ndof+1:))=ans(g(ndof+1:))+phi1
   END DO elements_3
!-----------------------apply loading increment---------------------------
   DO i=1,loaded_nodes; ans(nf(1:2,no(i)))=val(i,:)*al(j); END DO
!-----------------------equation solution---------------------------------
   CALL spabac_gauss(kv,ans,kdiag); loads=loads+ans; loads(0)=zero
   IF(j/npri*npri==j)WRITE(11,'(5E12.4)')time,tot_load,loads(nf(:,nres))
!-----------------------recover stresses at nip integrating points--------
   nip=1; DEALLOCATE(points,weights)
   ALLOCATE(points(nip,ndim),weights(nip))
   CALL sample(element,points,weights)
!   WRITE(11,'(A,I2,A)')" The integration point (nip=",nip,") stresses are:"
!   WRITE(11,'(A,A)')" Element x-coord     y-coord",                         &
!                    "     sig_x       sig_y       tau_xy"
   elements_4: DO iel=1,nels
     CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
     num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
     eld=loads(g(:ndof)); gauss_pts_2: DO i=1,nip
       CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
       gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
       deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
       sigma=MATMUL(dee,MATMUL(bee,eld))
!        IF(j/npri*npri==j)WRITE(11,'(I5,6E12.4)')iel,gc,sigma
     END DO gauss_pts_2
   END DO elements_4
 END DO time_steps
 CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p93
```

New scalar integers:

`j`	simple counter
`loaded_nodes`	number of loaded nodes
`ndof`	number of displacement degrees of freedom per element
`nlfp`	number of load function points
`nls`	maximum number of load steps
`npri`	output printed every `npri` time steps
`nprops`	number of material properties
`np_types`	number of different property types
`nres`	node number at which time history is to be printed
`nst`	number of stress/strain terms
`nstep`	number of time steps required

New scalar reals:

`dtim`	calculation time step
`theta`	time integration weighting parameter
`time`	holds time elapsed *t*
`tot_load`	accumulated load

New dynamic integer arrays:

`etype`	element property types
`kdiag`	diagonal term locations

New dynamic real arrays:

`al`	load steps at resolution of calculation time step
`ans`	rhs “load” increment vector
`bee`	strain-displacement matrix
`c`	coupling matrix
`dee`	stress–strain matrix
`eld`	element nodal displacements
`gc`	integrating point coordinates
`km`	element stiffness matrix
`kc`	element conductivity matrix
`kv`	global stiffness matrix
`lf`	input load/time function
`phi0`	used in element-by-element “gather” algorithm
`phi1`	used in element-by-element “scatter” algorithm
`sigma`	stress terms

`store_kc`	stores element `kc` matrices
`val`	nodal loads weighting factors
`vol`	related to the volumetric strain
`volf`	used to compute coupling matrix

The analysis of the behaviour of porous elastic solids (Biot 1941) under load is in many ways analogous to the coupled flow analysis described in the previous program. The displacements of the soil skeleton take over the role of the velocities u and v, and the excess porewater pressure (now called u_w) the role of the fluid pressure p (nodal freedoms are in the order u, v, u_w).

The differential equations to be solved are (2.136) and (2.137). Due to the coupling of fluid and solid phases there arises the complication that the applied "total" stresses, $\{\boldsymbol{\sigma}\}$, are divided between a portion carried by the soil skeleton, called *effective* stress $\{\boldsymbol{\sigma}'\}$, and a portion carried by the pore water, called in soil mechanics the *pore pressure* and denoted in Chapter 2 by $\{\mathbf{u}_w\}$.

After discretisation in space by finite elements, the coupled equations are given by (2.139). These can be seen to be partly algebraic equations and partly first order differential equations in time. In the incremental load method used here, discretisation in time by the θ-method leads to equations (3.115), which are in principle no different to (3.94) for uncoupled problems. If using an assembly approach, solutions will involve setting up the coupled global "stiffness" matrix on the left side of these equations (`kv`), followed by an equation solution for every time step to obtain the incremental solutions followed by an update of the variables from (3.116). For constant element properties and time step Δt, the left hand side matrix `kv` needs to be factorised only once, the remainder of the solution involving matrix-by-vector multiplication on the right hand side, followed by forward and back substitution.

Closer examination of equation (3.115) will reveal that some of the diagonal terms of the left hand side matrix will be negative, thus the usual Cholesky solution strategy will fail due to the need to take square roots. The subroutines `sparin_gauss` and `spabac_gauss` which operate on the skyline `kv` vector use Gaussian $[\mathbf{L}][\mathbf{D}][\mathbf{L}]^{\mathrm{T}}$ factorisation are therefore introduced for the first time (see Section 3.8). Other new subroutines include, `load_function`, which reads the input load-time function and linearly interpolates the function to produce a load-time function at the calculation time-step resolution held in `al`, and `formke` which forms the lhs element matrix `ke` from equation (3.115). The structure chart describing the algorithm is shown in Figure 9.7.

The problem chosen is of a plane strain "oedometer" specimen as shown by the mesh and input data given in Figure 9.8. The base and sides of the mesh are impermeable "no-flow" boundaries, and "smooth" roller boundary conditions are imposed on the sides. The top of the specimen is drained and subjected to a "ramp" loading of the form shown in Figure 9.9.

The first line of data reads the number of elements in each direction of the rectangular mesh (`nxe` and `nye`), and the number of property types (`np_types`). The properties are read next in the order k_x/γ_w, k_y/γ_w, E' and ν'. Since `np_types=1` in this homogeneous example, the `etype` data is not needed. Next comes the rectangular mesh coordinate data `x_coords` and `y_coords`, followed by the time stepping and output data. In this case, the data calls for `nstep=300` calculation steps, with a time step of `dtim=0.01`. The

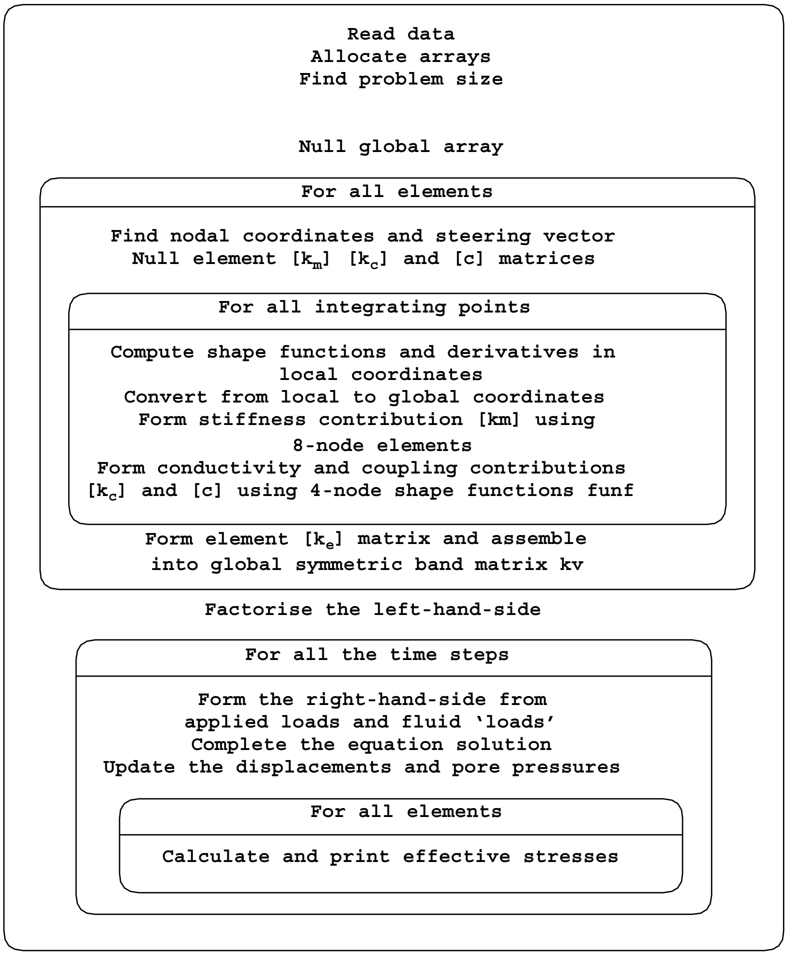
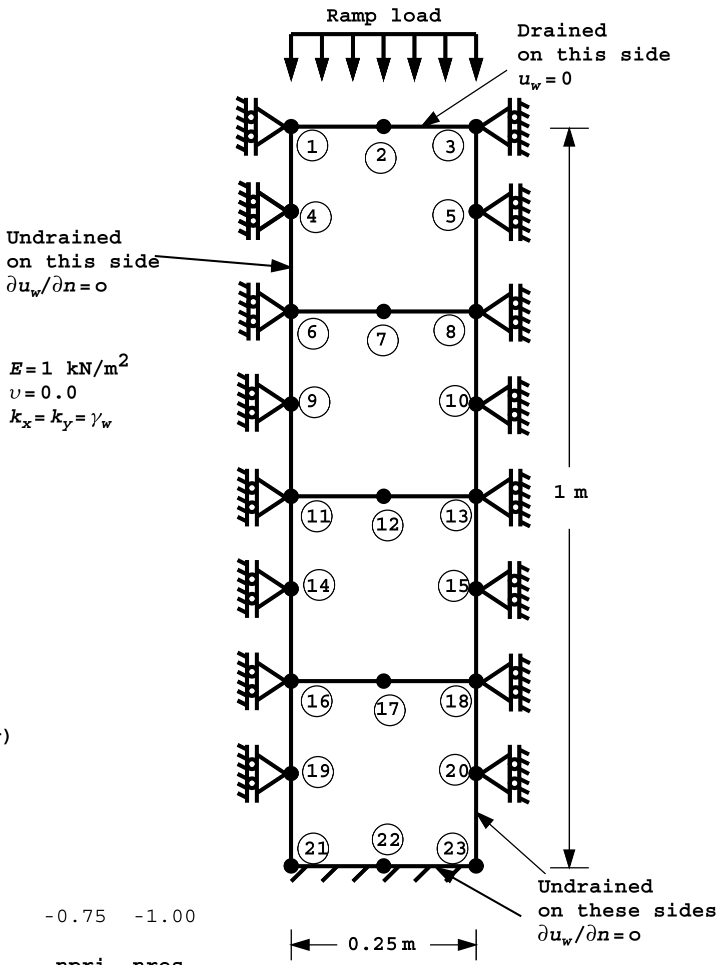

Read data
Allocate arrays
Find problem size
Null global array
For all elements
Find nodal coordinates and steering vector
Null element [k$_m$] [k$_c$] and [c] matrices
For all integrating points
Compute shape functions and derivatives in local coordinates
Convert from local to global coordinates
Form stiffness contribution [km] using 8-node elements
Form conductivity and coupling contributions [k$_c$] and [c] using 4-node shape functions funf
Form element [k$_e$] matrix and assemble into global symmetric band matrix kv
Factorise the left-hand-side
For all the time steps
Form the right-hand-side from applied loads and fluid 'loads'
Complete the equation solution
Update the displacements and pore pressures
For all elements
Calculate and print effective stresses

Figure 9.7 Structure chart for incremental form of Biot analysis with global matrix assembly in Program 9.3

time stepping parameter is set to `theta=0.5`. Displacement and pore pressure output is requested at node `nres=21` at every tenth time step `npri=10`. The nodal freedom data comes next, fixing the pore pressures at the top of the mesh to zero (drained) and also the x-displacements at each side of the mesh to zero (smooth). This implies "oedometer" conditions, with drainage at the top only. As with Programs 9.1 and 9.2, no pressures are computed at the mid-side nodes, so the third freedom at all mid-side nodes is removed from the analysis. The next data provide the load weightings corresponding to a unit pressure (Appendix A) applied to the 3 nodes at the top of the mesh. The final data define the load-time function to be applied at the top of the mesh. The bilinear function described in Figure 9.9 is defined by just three coordinates and is interpolated linearly at each calculation time step. The example given in Figure 9.8 is for a ramp load that reaches its maximum value of 1.0 after $t_o = 0.5$ seconds.


Ramp load
Drained on this side $u_w = 0$
Undrained on this side $\partial u_w/\partial n = 0$
$E = 1$ kN/m^2
$\upsilon = 0.0$
$k_x = k_y = \gamma_w$
1 m
Undrained on these sides $\partial u_w/\partial n = 0$
0.25 m
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23

```
nxe  nye  np_types
1     4     1

prop(kx/γw,ky/γw,e,v)
1.0  1.0  1.0  0.0

etype(not needed)

x_coords y_coords
0.0   0.25
0.0  -0.25  -0.50     -0.75  -1.00

dtim  nstep  theta     npri  nres
0.01   300    0.5       10    21

nr,(k,nf(:,k),i=1,nr)
23
 1 0 1 0   2 1 1 0   3 0 1 0   4 0 1 0        5 0 1 0
 6 0 1 1   7 1 1 0   8 0 1 1   9 0 1 0       10 0 1 0
11 0 1 1  12 1 1 0  13 0 1 1  14 0 1 0       15 0 1 0
16 0 1 1  17 1 1 0  18 0 1 1  19 0 1 0       20 0 1 0
21 0 0 1  22 0 0 0  23 0 0 1

loaded_nodes,(no(i),val(i,:),i=1,loaded_nodes)
3
1  0.0  -0.041667   2  0.0  -0.166667   3  0.0  -0.041667

nlfp,(lf(:,i),i=1,nlfp)
3
0.0 0.0  0.5 1.0  3.0 1.0
```

Figure 9.8 Mesh and data for Program 9.3 example

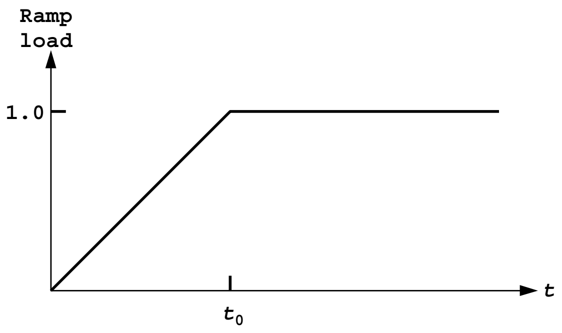


Figure 9.9 Ramp loading

```
There are   32 equations and the skyline storage is  280

Results at node   21
   time        load        x-disp      y-disp   porepressure
 0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
 0.1000E+00  0.2000E+00  0.0000E+00  0.0000E+00 -0.1994E+00
 0.2000E+00  0.4000E+00  0.0000E+00  0.0000E+00 -0.3743E+00
 0.3000E+00  0.6000E+00  0.0000E+00  0.0000E+00 -0.5125E+00
 0.4000E+00  0.8000E+00  0.0000E+00  0.0000E+00 -0.6203E+00
 0.5000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.7043E+00
 0.6000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.5703E+00
 0.7000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.4463E+00
 0.8000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.3478E+00
 0.9000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.2709E+00
 0.1000E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2110E+00
 0.1100E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1643E+00
 0.1200E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1280E+00
 0.1300E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.9968E-01
 0.1400E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.7764E-01
 0.1500E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.6047E-01
 0.1600E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.4709E-01
 0.1700E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.3668E-01
 0.1800E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2857E-01
 0.1900E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2225E-01
 0.2000E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1733E-01
 0.2100E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1350E-01
 0.2200E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1051E-01
 0.2300E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.8187E-02
 0.2400E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.6377E-02
 0.2500E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.4966E-02
 0.2600E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.3868E-02
 0.2700E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.3013E-02
 0.2800E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2346E-02
 0.2900E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1827E-02
 0.3000E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1423E-02
```

Figure 9.10 Results from Program 9.3 example

Turning to the program, the nodal coordinates and steering vector are again provided by subroutine `geom_rect` with numbering in the x-direction, and the subroutine `formke` builds up the `ke` matrix which is then assembled into the (symmetric) global skyline matrix `kv`.

The subroutine `sparin_gauss` is used to factorise the symmetric left hand side global matrix `kv` and the time stepping loop is entered. The right hand side vector is summed element-by-element from the fluid "loading" and the external load increment held in `al`. The subroutine `spabac_gauss` completes the solution and the element effective stresses can be recovered at the element Gauss points if required.

The results are shown as Figure 9.10, and the pore pressure at the base of the mesh (node 21) is plotted against time in Figure 9.11 for two different ramp rise-times, $t_o = 0.1$ and $t_o = 0.5$. The "time-factor" T in Figure 9.11 is the dimensionless number

$$T = \frac{c_v t}{D^2} \tag{9.1}$$

where D is the "maximum drainage path" of 1.0 in the present instance. The coefficient of consolidation c_v is found from,

$$c_v = \frac{k}{m_v \gamma_w} \tag{9.2}$$

where

$$m_v = \frac{(1+\nu')(1-2\nu')}{E'(1-\nu')} \tag{9.3}$$

k is the soil permeability (Chapter 7) and γ_w is the unit weight of water. In the present example $\nu' = 0$ and $E' = 1.0$, so $m_v = 1.0$. Similarly, $k/\gamma_w = 1.0$ so that $T = t$. The results are in very close agreement with Schiffman (1960), and problems of practical importance have been solved since (Smith and Hobbs, 1976).

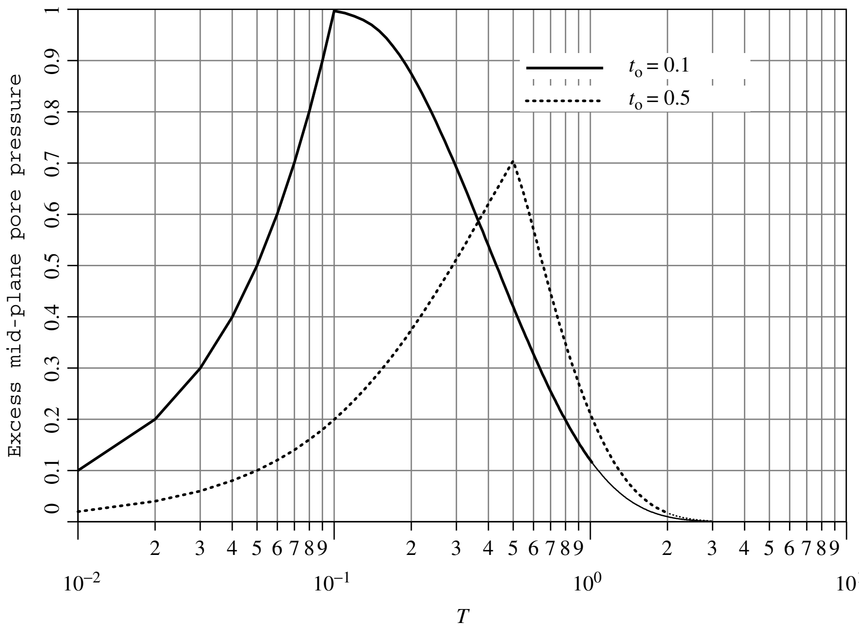


Figure 9.11 Mid-plane pore pressure response to ramp loading from Program 9.3

Program 9.4 Plane strain consolidation analysis of a Biot poro-elastic-plastic (Mohr–Coulomb) material using 8-node rectangular quadrilaterals for displacements coupled to 4-node rectangular quadrilaterals for pressures. Freedoms numbered in the order u-v-u_w. Incremental version. Viscoplastic strain method.

```
PROGRAM p94
!-------------------------------------------------------------------------
! Program 9.4 Plane strain consolidation analysis of a Biot elastic-plastic
!             (Mohr-Coulomb) material using 8-node rectangular
!             quadrilaterals for displacements coupled to 4-node
!             rectangular quadrilaterals for pressures. Incremental
!             version. Viscoplastic strain method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,iters,j,k,limit,loaded_nodes,ndim=2,ndof=16,nels,neq,     &
   nip=4,nlfp,nls,nn,nod=8,nodf=4,nodof=3,npri,nprops=7,np_types,nr,nres, &
   nst=4,nstep,ntot=20,nxe,nye
 REAL(iwp)::coh,cons,ddt,det,dpore,dq1,dq2,dq3,dsbar,dt,dtim,d4=4.0_iwp,  &
   d180=180.0_iwp,e,f,lode_theta,one=1.0_iwp,phi,pi,psi,sigm,snph,        &
   start_dt=1.e15_iwp,theta,time,tol,tot_load,two=2.0_iwp,v,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::nf(:,:),g(:),num(:),g_num(:,:),g_g(:,:),etype(:),   &
   kdiag(:),no(:)
 REAL(iwp),ALLOCATABLE::al(:),ans(:),bee(:,:),bdylds(:),bload(:),c(:,:),  &
   coord(:,:),dee(:,:),der(:,:),derf(:,:),deriv(:,:),derivf(:,:),devp(:), &
   disps(:),eld(:),eload(:),eps(:),erate(:),evp(:),evpt(:,:,:),flow(:,:), &
   funf(:),g_coord(:,:),jac(:,:),kay(:,:),ke(:,:),km(:,:),kp(:,:),kv(:),  &
   lf(:,:),loads(:),m1(:,:),m2(:,:),m3(:,:),newdis(:),oldis(:),phi0(:),   &
   phi1(:),points(:,:),prop(:,:),sigma(:),store_kp(:,:,:),stress(:),      &
   tensor(:,:,:),val(:,:),vol(:),volf(:,:),weights(:),x_coords(:),        &
   y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(dee(nst,nst),points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),&
   jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),           &
   derf(ndim,nodf),funf(nodf),bee(nst,ndof),km(ndof,ndof),eld(ndof),      &
   sigma(nst),kp(nodf,nodf),g_g(ntot,nels),ke(ntot,ntot),c(ndof,nodf),    &
   x_coords(nxe+1),phi0(nodf),y_coords(nye+1),vol(ndof),nf(nodof,nn),     &
   volf(ndof,nodf),g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip),&
   phi1(nodf),store_kp(nodf,nodf,nels),tensor(nst+1,nip,nels),eps(nst),   &
   evp(nst),evpt(nst,nip,nels),bload(ndof),eload(ndof),erate(nst),g(ntot),&
   devp(nst),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),           &
   stress(nst),etype(nels),prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)cons,x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),loads(0:neq),ans(0:neq),bdylds(0:neq),disps(0:neq),  &
   newdis(0:neq),oldis(0:neq))
 READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(no(i),val(i,:),i=1,loaded_nodes); READ(10,*)tol,limit,nlfp
 ALLOCATE(lf(2,nlfp)); READ(10,*)lf; nls=FLOOR(lf(1,nlfp)/dtim)
 IF(nstep>nls)nstep=nls; ALLOCATE(al(nstep))
 CALL load_function(lf,dtim,al); kdiag=0
!-----------------------loop the elements to find global arrays sizes-----
 elements_1: DO iel=1,nels
```

```
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   g(1:15:2)=nf(1,num(:)); g(2:16:2)=nf(2,num(:)); g(17:)=nf(3,num(1:7:2))
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                   &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 loads=zero; disps=zero; tensor=zero
 CALL sample(element,points,weights); kv=zero
!-----------------------global matrix assembly----------------------------
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; c=zero; kp=zero
   gauss_points_1: DO i=1,nip
!-----------------------elastic solid contribution------------------------
     CALL shape_der(der,points,i); jac=MATMUL(der,coord)
     det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
     tensor(1:2,i,iel)=cons; tensor(4,i,iel)=cons; tensor(5,i,iel)=zero
     CALL beemat(bee,deriv); vol(:)=bee(1,:)+bee(2,:)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----------------------fluid contribution--------------------------------
     CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
     derivf=MATMUL(jac,derf)
     kp=kp+MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtim
     DO k=1,nodf; volf(:,k)=vol(:)*funf(k); END DO; c=c+volf*det*weights(i)
   END DO gauss_points_1; store_kp(:,:,iel)=kp
   CALL formke(km,kp,c,ke,theta); CALL fsparv(kv,ke,g,kdiag)
 END DO elements_2
!-----------------------factorise equations-------------------------------
 CALL sparin_gauss(kv,kdiag)
!-------------------------------------------------------------------------
 pi=ACOS(-one); dt=start_dt
 DO i=1,np_types
   phi=prop(5,i); snph=SIN(phi*pi/d180); e=prop(3,i); v=prop(4,i)
   ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
 END DO; bdylds=zero; evpt=zero; oldis=zero; time=zero
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/A,I5)')" Results at node",nres
 WRITE(11,'(A)')                                                         &
   "    time        load        x-disp      y-disp   porepressure iters"
 WRITE(11,'(5E12.4)')0.0,0.0,0.0,0.0,0.0
 time_steps: DO j=1,nstep
   time=time+dtim; tot_load=SUM(al(1:j)); ans=zero; bdylds=zero
   evpt=zero; newdis=zero
 elements_3: DO iel=1,nels
     g=g_g(:,iel); kp=store_kp(:,:,iel)
     phi0=disps(g(ndof+1:)); phi1=MATMUL(kp,phi0)
     ans(g(ndof+1:))=ans(g(ndof+1:))+ phi1; ans(0)=zero
   END DO elements_3
!-----------------------apply loading increment---------------------------
   DO i=1,loaded_nodes; ans(nf(1:2,no(i)))=val(i,:)*al(j); END DO; iters=0
!-----------------------plastic iteration loop----------------------------
   its: DO
     iters=iters+1
     WRITE(*,'(A,I6,A,I4)')" time step",j,"  iteration",iters
     loads=ans+bdylds; CALL spabac_gauss(kv,loads,kdiag)
```

```
!-----------------------check plastic convergence------------------------
     newdis=loads; newdis(nf(3,:))=zero
     CALL checon(newdis,oldis,tol,converged); IF(iters==1)converged=.FALSE.
     IF(converged.OR.iters==limit)bdylds=zero
!-----------------------go round the Gauss Points -----------------------
     elements_4: DO iel=1,nels
       phi=prop(5,etype(iel)); coh=prop(6,etype(iel))
       psi=prop(7,etype(iel))
       CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
       num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
       eld=loads(g(1:ndof)); bload=zero
       gauss_points_2: DO i=1,nip
         CALL shape_der(der,points,i)
         jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
         deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
         eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel); sigma=MATMUL(dee,eps)
         stress=sigma+tensor(1:4,i,iel)
         CALL invar(stress,sigm,dsbar,lode_theta)
!-----------------------check whether yield is violated-------------------
         CALL mocouf(phi,coh,sigm,dsbar,lode_theta,f)
         IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
           IF(f>=zero)THEN
             CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
             CALL formm(stress,m1,m2,m3)
             flow=f*(m1*dq1+m2*dq2+m3*dq3); erate=MATMUL(flow,stress)
             evp=erate*dt; evpt(:,i,iel)=evpt(:,i,iel)+evp
             devp=MATMUL(dee,evp)
           END IF
         END IF
         IF(f>=zero)THEN; eload=MATMUL(TRANSPOSE(bee),devp)
           bload=bload+eload*det*weights(i)
         END IF
         IF(converged.OR.iters==limit)THEN
!--------------update the Gauss Point stresses and porepressures----------
           tensor(1:4,i,iel)=stress; dpore=zero
           CALL shape_fun(funf,points,i)
           DO k=1,nodf; dpore=dpore+funf(k)*loads(g(k+ndof)); END DO
           tensor(5,i,iel)=tensor(5,i,iel)+dpore
         END IF
       END DO gauss_points_2
!-----------------------compute the total bodyloads vector ---------------
       bdylds(g(1:ndof))=bdylds(g(1:ndof))+bload; bdylds(0)=zero
     END DO elements_4; IF(converged.OR.iters==limit)EXIT
   END DO its; disps=disps+loads
   IF(j/npri*npri==j.OR.iters==limit)WRITE(11,'(5E12.4,I5)')            &
     time,tot_load,disps(nf(:,nres)),iters
   IF(iters==limit)EXIT
 END DO time_steps
 CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,13)
 CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,14)
STOP
END PROGRAM p94
```

New scalar reals:

`coh`	soil cohesion
`cons`	consolidating stress (σ_3)
`ddt`	used to find the critical time step

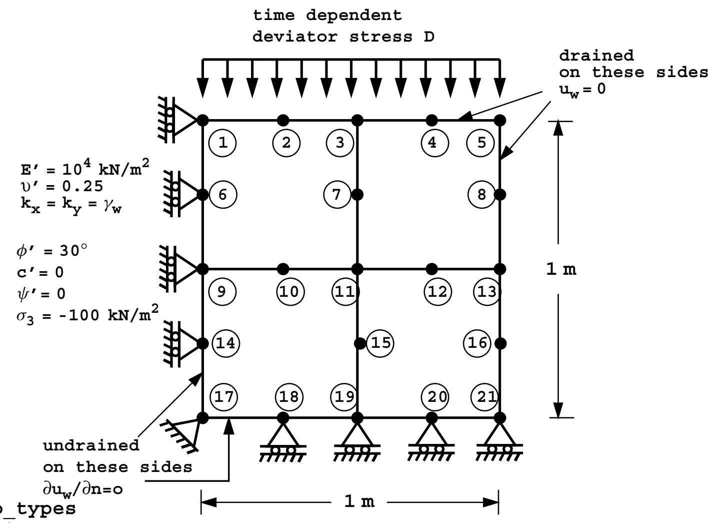
`dpore`	holds the accumulated pore pressure
`dq1`	plastic potential derivative, $\partial Q/\partial\sigma_m$
`dq2`	plastic potential derivative, $\partial Q/\partial J_2$
`dq3`	plastic potential derivative, $\partial Q/\partial J_3$
`dsbar`	invariant, $\overline{\sigma}$
`dt`	critical viscoplastic time step
`d4`	set to 4.0
`d180`	set to 180.0
`e`	Young's Modulus
`f`	yield function
`load_theta`	Lode angle, θ
`phi`	friction angle (degrees)
`pi`	set to π
`psi`	dilation angle (degrees)
`sigm`	mean stress, σ_m
`snph`	sin of `phi`
`start_dt`	starting value of `dt`
`tol`	plastic convergence tolerance
`v`	Poisson's ratio

New dynamic real arrays:

`bdylds`	self-equilibrating global body loads
`bload`	self-equilibrating element body loads
`devp`	product $[\mathbf{D}^e]\{\boldsymbol{\Delta\epsilon}^{vp}\}$
`disps`	global displacements and pore pressures
`eload`	integrating point contribution to `bload`
`eps`	strain terms
`erate`	viscoplastic strain rate, $\left\{\dot{\boldsymbol{\epsilon}}^{vp}\right\}$
`evp`	viscoplastic strain rate increment, $\{\boldsymbol{\delta\epsilon}^{vp}\}$
`evpt`	holds running total of viscoplastic strains, $\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}$
`flow`	holds $\{\partial Q/\partial\boldsymbol{\sigma}\}$
`m1`	used to compute $\{\partial\sigma_m/\partial\boldsymbol{\sigma}\}$
`m2`	used to compute $\{\partial J_2/\partial\boldsymbol{\sigma}\}$
`m3`	used to compute $\{\partial J_3/\partial\boldsymbol{\sigma}\}$
`newdis`	"new" displacements and pore pressures
`oldis`	"old" displacements and pore pressures
`stress`	stress terms
`tensor`	holds running total of all integrating point stress terms

For non-linear problems involving elastic–plastic solid skeletons in a Biot analysis, the advantage of an incremental form of the equations, which was not strictly necessary for an elastic material, becomes apparent. The resulting program is an amalgamation of Programs 9.3 and 6.3 which used the viscoplastic strain method for redistributing excess internal stresses (Griffiths, 1994).

The three-dimensional array `tensor` is used to store the element integrating point stresses, with the four effective stress components coming first, followed by the pore water pressure as the fifth component.



```
nxe   nye   np_types
2     2      1

prop(kx/γw,ky/γw,e,v,phi,c,psi)
1.0e-6  1.0e-6  1.0e4  0.25  30.0  0.0  0.0

etype(not needed)

cons
-100.0

x_coords, y_coords
0.0   0.5   1.0
0.0  -0.5  -1.0

dtim  nstep  theta  npri  nres
0.5    200    0.5     1    1

nr,(k,nf(:,k),i=1,nr)
20
 1 0 1 0   2 1 1 0   3 1 1 0   4 1 1 0   5 1 1 0
 6 0 1 0   7 1 1 0   8 1 1 0   9 0 1 1  10 1 1 0
12 1 1 0  13 1 1 0  14 0 1 0  15 1 1 0  16 1 1 0
17 0 0 1  18 1 0 0  19 1 0 1  20 1 0 0  21 1 0 0

loaded_nodes,(no(i),val(i,:),i=1,loaded_nodes)
5
1  0.0  -0.08333   2  0.0  -0.33333   3  0.0  -0.16667
4  0.0  -0.33333   5  0.0  -0.08333

tol   limit
0.001  250

nlfp,(lf(:,i),i=1,nlfp)
2
0.0 0.0  10.0 150.0
```

Figure 9.12 Mesh and data for Program 9.4 example

The illustrative problem shown in Figure 9.12 involves compression of a plane strain block of saturated elastic–plastic cohesionless soil by a time-dependent “deviator” stress D which is the difference between the vertical and (constant) horizontal stresses on the soil.

The data follows a similar course to that followed for Program 9.3. The number of properties has expanded to seven (`nprops=7`) with the addition of the friction angle ϕ', the cohesion c' and the dilation angle ψ. The "permeability" property k/γ_w of the soil in this example is isotropic and set to 1×10^{-6}. The current example models a cohesionless soil with $\phi' = 30^\circ$ and no dilation. The soil is initially consolidated to an isotropic compressive stress (σ_3) of -100 kN/m^2 read as `cons`. The coordinate data for `x_coords` and `y_coords` is followed by the familiar time stepping and output control parameters. The current example calls for `nstep=200` calculation time steps of `dtim=0.5` secs with `theta=0.5` (Crank–Nicolson). Output is required every time step (`npri=1`) at node `nres=1`. The `nr` data indicate 20 restrained nodes, which include rollers on the left and bottom boundaries, drainage conditions on the top and right boundaries and removal of the third freedom at all mid-side nodes, as described in Program 9.3. The next data provide the load weightings corresponding to a unit pressure (Appendix A), to be applied to the 5 nodes at the top of the mesh. The next line reads the tolerance and iteration ceiling (`tol` and `limit`) for plastic iterations, as was first used in Program 4.5 and again extensively in Chapter 6. The final data define the load-time function to be applied at the top of the mesh. In this example the deviator stress is to increase linearly with time at a "fast" rate given by $\mathrm{d}D/\mathrm{d}t = 15$ kNm^{-2} s^{-1}, so just two load function coordinates are required.

The results from this analysis are listed as Figure 9.13 and plotted in Figure 9.14, together with the results of a second analysis in which a "slow" loading rate of $\mathrm{d}D/\mathrm{d}t = 0.02$ kNm^{-2} s^{-1} was applied. The deviator stress at failure D_f was computed to be about 100 kN/m^2 for the "fast" loading rate and 200 kN/m^2 for the "slow" loading rate.

For plane strain compression of a non-dilative saturated soil, Griffiths(1985) produced the solution,

$$D_\mathrm{f} = \frac{\sigma_3(K_p - 1)(2\beta_{ps} + 1)}{(K_p + 1)\beta_{ps} + 1} \tag{9.4}$$

in which $\beta_{ps} \to \infty$ and $\beta_{ps} = 0$ give undrained and drained limiting conditions respectively.

In this example, $K_p = \tan^2(45^\circ + \phi'/2) = 3$, so for a consolidating stress of $\sigma_3 = 100$ kN/m^2, (9.4) with $\beta_{ps} \to \infty$ gives $D_\mathrm{f} \approx 100$ kN/m^2, indicating that "fast" loading in this case is giving essentially "undrained" conditions. The "slow" loading result of

```
There are   36 equations and the skyline storage is  466

Results at node    1
   time        load        x-disp      y-disp     porepressure iters
 0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
 0.5000E+00  0.7500E+01  0.0000E+00 -0.5025E-03  0.0000E+00    2
 0.1000E+01  0.1500E+02  0.0000E+00 -0.1009E-02  0.0000E+00    2
 0.1500E+01  0.2250E+02  0.0000E+00 -0.1520E-02  0.0000E+00    2
 0.2000E+01  0.3000E+02  0.0000E+00 -0.2035E-02  0.0000E+00    2
 0.2500E+01  0.3750E+02  0.0000E+00 -0.2555E-02  0.0000E+00    2
 0.3000E+01  0.4500E+02  0.0000E+00 -0.3078E-02  0.0000E+00    2
 0.3500E+01  0.5250E+02  0.0000E+00 -0.3606E-02  0.0000E+00    2
 0.4000E+01  0.6000E+02  0.0000E+00 -0.4138E-02  0.0000E+00    2
 0.4500E+01  0.6750E+02  0.0000E+00 -0.4674E-02  0.0000E+00    2
 0.5000E+01  0.7500E+02  0.0000E+00 -0.5213E-02  0.0000E+00    2
 0.5500E+01  0.8250E+02  0.0000E+00 -0.5757E-02  0.0000E+00    2
 0.6000E+01  0.9000E+02  0.0000E+00 -0.6304E-02  0.0000E+00    2
 0.6500E+01  0.9750E+02  0.0000E+00 -0.6886E-02  0.0000E+00    9
 0.7000E+01  0.1050E+03  0.0000E+00 -0.4326E-01  0.0000E+00  250
```

Figure 9.13 Results from Program 9.4 example with $\mathrm{d}D/\mathrm{d}t = 15$

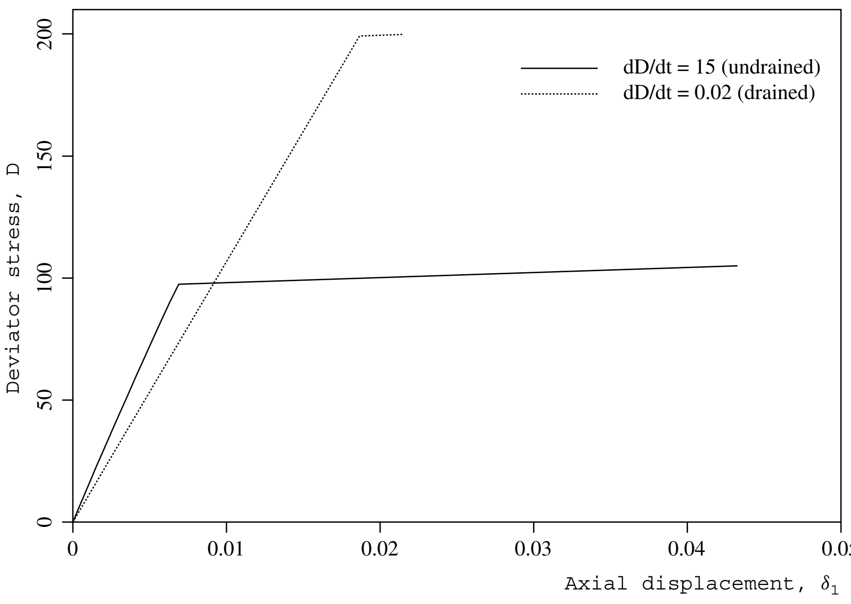


Figure 9.14 Response for different loading rates from Program 9.4

$D_\mathrm{f} \approx 200$ kN/m^2 corresponds essentially to the classical drained solution, also given by (9.4) with $\beta_{ps} = 0$.

Program 9.5 Plane strain consolidation analysis of a Biot poro-elastic solid using 8-node rectangular quadrilaterals for displacements coupled to 4-node rectangular quadrilaterals for pressures. Freedoms numbered in the order u-v-u_w. Absolute load version. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver.

```
PROGRAM p95
!-------------------------------------------------------------------------
! Program 9.5 Plane strain consolidation analysis of a Biot elastic
!             solid using 8-node rectangular quadrilaterals for
!             displacements coupled to 4-node rectangular quadrilaterals
!             for pressures. Absolute load version.
!             No global stiffness matrix assembly.
!             Diagonally preconditioned conjugate gradient solver.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,i,iel,j,k,loaded_nodes,ndim=2,ndof=16,nels,   &
   neq,nip=4,nlfp,nls,nn,nod=8,nodf=4,nodof=3,npri,nprops=4,np_types,nr,  &
   nres,nst=3,nstep,ntot=20,nxe,nye
 REAL(iwp)::alpha,beta,cg_tol,det,dtim,one=1.0_iwp,theta,time,tot_load,up,&
   zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::cg_converged
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),nf(:,:),no(:),    &
   num(:)
 REAL(iwp),ALLOCATABLE::al(:),ans(:),bee(:,:),c(:,:),coord(:,:),d(:),     &
   dee(:,:),der(:,:),derf(:,:),deriv(:,:),derivf(:,:),diag_precon(:),     &
   eld(:),fun(:),funf(:),gc(:),g_coord(:,:),jac(:,:),kay(:,:),kd(:,:),    &
   ke(:,:),km(:,:),kc(:,:),lf(:,:),loads(:),p(:),points(:,:),prop(:,:),   &
   sigma(:),storkd(:,:,:),storke(:,:,:),u(:),val(:,:),vol(:),volf(:,:),   &
   weights(:),x(:),xnew(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,cg_tol,cg_limit,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(dee(nst,nst),points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),&
   jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),           &
   derf(ndim,nodf),funf(nodf),bee(nst,ndof),km(ndof,ndof),kc(nodf,nodf),  &
   g_g(ntot,nels),ke(ntot,ntot),kd(ntot,ntot),c(ndof,nodf),fun(nod),      &
   x_coords(nxe+1),y_coords(nye+1),vol(ndof),nf(nodof,nn),g(ntot),        &
   volf(ndof,nodf),g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip),&
   storke(ntot,ntot,nels),storkd(ntot,ntot,nels),etype(nels),             &
   prop(nprops,np_types),gc(ndim),sigma(nst),eld(ndof))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(loads(0:neq),ans(0:neq),p(0:neq),x(0:neq),xnew(0:neq),u(0:neq), &
   diag_precon(0:neq),d(0:neq))
 READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(no(i),val(i,:),i=1,loaded_nodes)
 READ(10,*)nlfp; ALLOCATE(lf(2,nlfp))
 READ(10,*)lf; nls=FLOOR(lf(1,nlfp)/dtim); IF(nstep>nls)nstep=nls
 ALLOCATE(al(nstep)); CALL load_function(lf,dtim,al)
!-----------------------loop the elements to set up element data----------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
   g(1:15:2)=nf(1,num(:)); g(2:16:2)=nf(2,num(:)); g(17:)=nf(3,num(1:7:2))
   g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 WRITE(11,'(A,I5,A)')" There are",neq," equations"
 CALL sample(element,points,weights); diag_precon=zero
!----------element matrix integration, storage and preconditioner---------
 elements_2: DO iel=1,nels
   kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
   CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel))); num=g_num(:,iel)
   coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; c=zero; kc=zero
   gauss_points_1: DO i=1,nip
!-----------------------elastic solid contribution------------------------
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     vol(:)=bee(1,:)+bee(2,:)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!----------------------fluid contribution---------------------------------
     CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
     derivf=MATMUL(jac,derf)
```

```
    kc=kc+MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtim
    CALL cross_product(vol,funf,volf); c=c+volf*det*weights(i)
  END DO gauss_points_1; CALL fmkdke(km,kc,c,ke,kd,theta)
  storke(:,:,iel)=ke; storkd(:,:,iel)=kd
  DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+theta*km(k,k); END DO
  DO k=1,nodf
    diag_precon(g(ndof+k))=diag_precon(g(ndof+k))-theta*theta*kc(k,k)
  END DO
 END DO elements_2
 diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/A,I5)')" Results at node",nres; loads=zero
 WRITE(11,'(4X,A)')                                                      &
   "time       load       x-disp      y-disp   porepressure  cg iters"
 WRITE(11,'(5E12.4)')0.0,0.0,loads(nf(:,nres))
 tot_load=zero; time=zero
 time_steps: DO j=1,nstep
   ans=zero; time=time+dtim
   elements_3: DO iel=1,nels
     g=g_g(:,iel); kd=storkd(:,:,iel); ans(g)=ans(g)+MATMUL(kd,loads(g))
   END DO elements_3; ans(0)=zero
!-----------------------apply absolute loading----------------------------
   DO i=1,loaded_nodes
     ans(nf(1:2,no(i)))=ans(nf(1:2,no(i)))+val(i,:)*(tot_load+theta*al(j))
   END DO
   tot_load=tot_load+al(j); d=diag_precon*ans; p=d; x=zero; cg_iters=0
!-----------------------pcg equation solution-----------------------------
   pcg: DO
     cg_iters=cg_iters+1; u=zero
     elements_4: DO iel=1,nels
       g=g_g(:,iel); ke=storke(:,:,iel); u(g)=u(g)+MATMUL(ke,p(g))
     END DO elements_4
     up=DOT_PRODUCT(ans,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
     ans=ans-u*alpha; d=diag_precon*ans; beta=DOT_PRODUCT(ans,d)/up
     p=d+p*beta; CALL checon(xnew,x,cg_tol,cg_converged)
     IF(cg_converged.OR.cg_iters==cg_limit)EXIT
   END DO pcg; loads=xnew; loads(0)=zero
   IF(j/npri*npri==j)WRITE(11,'(5E12.4,I7)')                             &
     time,tot_load,loads(nf(:,nres)),cg_iters
!-----------------------recover stresses at nip integrating points--------
   nip=1; DEALLOCATE(points,weights)
   ALLOCATE(points(nip,ndim),weights(nip))
   CALL sample(element,points,weights)
!   WRITE(11,'(A,I2,A)')" The integration point (nip=",nip,") stresses are:"
!   WRITE(11,'(A,A)')" Element x-coord     y-coord",                        &
!                    "     sig_x       sig_y       tau_xy"
   elements_5: DO iel=1,nels
     CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
     num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
     eld=loads(g(:ndof))
     gauss_pts_2: DO i=1,nip
       CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
       gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
       deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
       sigma=MATMUL(dee,MATMUL(bee,eld))
```

```
!       IF(j/npri*npri==j)WRITE(11,'(I5,6E12.4)')iel,gc,sigma
     END DO gauss_pts_2
   END DO elements_5
 END DO time_steps
STOP
END PROGRAM p95
```

New scalar reals:

`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\,\{\mathbf{R}\}_k$ from (3.22)

New dynamic real arrays:

`d`	vector used in (3.22)
`diag_precon`	diagonal preconditioner vector
`kd`	right hand side element matrix from "Biot" analysis
`storkd`	stores augmented diagonal terms
`x`	"old" solution vector
`xnew`	"new" solution vector

For the final program of this chapter, we return to the analysis of a poro-elastic "Biot" material, but this time using a "mesh free" approach with a pcg solver that involves no global matrix assembly. The absolute loading version is demonstrated this time as described by equation (3.112), in which each time step involves matrix–vector multiplication followed by equation solution. The process may be written as,

$$[\mathbf{k}_e]\,\{\boldsymbol{\phi}\}_1 = [\mathbf{k}_d]\,\{\boldsymbol{\phi}\}_0 + \{\mathbf{f}\} \tag{9.5}$$

which can all be performed at the element level using the pcg algorithm. The subroutine `fmkdke` forms the (symmetric) $[\mathbf{k}_e]$ matrix (`ke`) on the left hand side and the (unsymmetric) $[\mathbf{k}_d]$ matrix (`kd`) to the right. The vectors $\{\boldsymbol{\phi}\}_0$ and $\{\boldsymbol{\phi}\}_1$, called `phi0` and `phi1` in programming terminology, represent the element displacements and excess pore pressures at the "old" and "new" time steps.

Inspection of equation (3.112) shows that the left hand side matrix $[\mathbf{k}_e]$ in this analyses is not positive definite due to the negative $[\mathbf{k}_c]$ terms. Simple diagonal preconditioning does yield a symmetric positive definite preconditioned matrix however, and this is the method used in Program 9.5. It is recognised that alternative preconditioning and iterative strategies may well be more efficient. The `ke` and `kd` element matrices are stored as `storke` and `storkd` for use later in the generation of right hand side "loading" and pcg solution iterations.

The problem solved is the same as was solved by Program 9.3, and data are listed as Figure 9.15. The extra data items are just the conjugate gradient iteration tolerance and iteration limit, `cg_tol` and `cg_limit` respectively. The results are listed as Figure 9.16 which are essentially identical to those in Figure 9.10. The output in Figure 9.16 indicates that approximately 19 pcg iterations were needed at each calculation time step.

```
nxe  nye  cg_tol  cg_limit  np_types
1     4   1.0e-5    200       1

prop(kx/γw,ky/γw,e,v)
1.0  1.0  1.0  0.0

etype(not needed)

x_coords y_coords
0.0   0.25
0.0  -0.25  -0.50  -0.75  -1.00

dtim  nstep  theta  npri  nres
0.01   300    0.5    10    21

nr,(k,nf(:,k),i=1,nr)
23
 1 0 1 0   2 1 1 0   3 0 1 0   4 0 1 0   5 0 1 0
 6 0 1 1   7 1 1 0   8 0 1 1   9 0 1 0  10 0 1 0
11 0 1 1  12 1 1 0  13 0 1 1  14 0 1 0  15 0 1 0
16 0 1 1  17 1 1 0  18 0 1 1  19 0 1 0  20 0 1 0
21 0 0 1  22 0 0 0  23 0 0 1

loaded_nodes,(no(i),val(i,:),i=1,loaded_nodes)
3
1  0.0  -0.041667   2  0.0  -0.166667   3  0.0  -0.041667

nlfp,(lf(:,i),i=1,nlfp)
3
0.0 0.0  0.5 1.0  3.0 1.0
```

Figure 9.15 Data for Program 9.5 example

```
There are   32 equations

 Results at node   21
    time        load        x-disp      y-disp   porepressure  cg iters
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.1000E+00  0.2000E+00  0.0000E+00  0.0000E+00 -0.1994E+00     17
  0.2000E+00  0.4000E+00  0.0000E+00  0.0000E+00 -0.3743E+00     19
  0.3000E+00  0.6000E+00  0.0000E+00  0.0000E+00 -0.5125E+00     19
  0.4000E+00  0.8000E+00  0.0000E+00  0.0000E+00 -0.6203E+00     19
  0.5000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.7043E+00     19
  0.6000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.5703E+00     19
  0.7000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.4463E+00     19
  0.8000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.3478E+00     19
  0.9000E+00  0.1000E+01  0.0000E+00  0.0000E+00 -0.2709E+00     19
  0.1000E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2110E+00     21
  0.1100E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1643E+00     19
  0.1200E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1280E+00     21
  0.1300E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.9968E-01     19
  0.1400E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.7764E-01     19
  0.1500E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.6047E-01     19
  0.1600E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.4710E-01     19
  0.1700E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.3668E-01     19
  0.1800E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2857E-01     19
  0.1900E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2225E-01     19
  0.2000E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1733E-01     19
  0.2100E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1349E-01     19
  0.2200E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1051E-01     19
  0.2300E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.8186E-02     19
  0.2400E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.6375E-02     19
  0.2500E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.4970E-02     19
  0.2600E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.3872E-02     19
  0.2700E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.3015E-02     21
  0.2800E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.2348E-02     19
  0.2900E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1829E-02     19
  0.3000E+01  0.1000E+01  0.0000E+00  0.0000E+00 -0.1425E-02     19
```

Figure 9.16 Results from Program 9.5 example

Glossary of variable names used in Chapter 9

Scalar integers:

`cg_iters`	pcg or BiCGStab iteration counter
`cg_limit`	pcg or BiCGStab iteration ceiling
`cg_tot`	total number of BiCGStab iterations
`ell`	BiCGStab parameter; taken as 4 in this example
`fixed_freedoms`	number of fixed freedoms
`i`	simple counter
`iel`	simple counter
`iters`	iteration counter
`iwp`	`SELECTED_REAL_KIND(15)`
`j`	simple counter
`k`	simple counter
`limit`	iteration ceiling
`loaded_nodes`	number of loaded nodes
`nband`	full bandwidth of non-symmetric matrix
`ndim`	number of dimensions
`ndof`	number of displacement degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nlfp`	number of load function points
`nls`	maximum number of load steps
`nip`	number of integrating points
`nn`	number of nodes in the mesh
`nod`	number of nodes per solid element
`nodf`	number of nodes per fluid element
`nodof`	number of degrees of freedom per node
`npri`	output printed every `npri` time steps
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes
`nres`	node number at which time history is to be printed
`nst`	number of stress/strain terms
`nstep`	number of time steps required
`ntot`	total number of degrees of freedom per element
`nxe`	number of elements in x-direction
`nye`	number of elements in y-direction

Scalar reals:

`alpha`	local variable
`beta`	local variable
`cg_tol`	BiCGStab convergence tolerance
`coh`	soil cohesion
`cons`	consolidating stress (σ_3)
`ddt`	used to find the critical time step
`det`	determinant of the Jacobian matrix

`dpore`	holds the accumulated pore pressure
`dq1`	plastic potential derivative, $\partial Q/\partial \sigma_m$
`dq2`	plastic potential derivative, $\partial Q/\partial J_2$
`dq3`	plastic potential derivative, $\partial Q/\partial J_3$
`dsbar`	invariant, $\overline{\sigma}$
`dt`	critical viscoplastic time step
`dtim`	calculation time step
`d4`	set to 4.0
`d180`	set to 180.0
`e`	Young's Modulus
`error`	measure of error in BiCGStab
`f`	yield function
`gama`	local variable
`kappa`	BiCGStab parameter; taken as zero in this example
`load_theta`	Lode angle, θ
`norm_r`	residual norm
`omega`	local variable
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`phi`	friction angle (degrees)
`pi`	set to π
`psi`	dilation angle (degrees)
`pt5`	set to 0.5
`rho`	fluid density
`rho1`	local variable
`r0_norm`	initial residual norm
`sigm`	mean stress, σ_m
`snph`	sin of `phi`
`start_dt`	starting value of `dt`
`theta`	time integration weighting parameter
`time`	holds time elapsed t
`tol`	convergence tolerance
`tot_load`	accumulated load at time t
`ubar`	average x-velocity
`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\{\mathbf{R}\}_k$ from (3.22)
`v`	Poisson's ratio
`x0`	initialisation value
`vbar`	average y-velocity
`visc`	molecular viscosity
`zero`	set to 0.0

Character variables:

`element`	element type

Scalar logicals:

`converged`	set to `.TRUE.` if solution converged
`cg_converged`	set to `.TRUE` if BiCGStab has converged

Dynamic integer arrays:

`etype`	element property types
`g`	element “steering” vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`kdiag`	diagonal term locations
`nf`	nodal freedoms
`no`	freedoms to be fixed
`node`	nodes with fixed values
`num`	element node numbers
`sense`	sense of freedom to be fixed

Dynamic real arrays:

`al`	load steps at resolution of calculation time step
`ans`	rhs “load” increment vector
`b`	right-hand side vector
`bdylds`	self-equilibrating global body loads
`bee`	strain-displacement matrix
`bload`	self-equilibrating element body loads
`c`	coupling matrix
`coord`	solid element nodal coordinates
`coordf`	fluid element nodal coordinates
`c11`	element submatrix (2.115)
`c12`	element submatrix (2.115)
`c21`	element submatrix (2.115)
`c23`	element submatrix (2.115)
`c32`	element submatrix (2.115)
`d`	vector used in (3.22)
`dee`	stress–strain matrix
`der`	solid shape function derivatives wrt local coordinates
`derf`	fluid shape function derivatives wrt local coordinates
`deriv`	solid shape function derivatives wrt global coordinates
`derivf`	fluid shape function derivatives wrt global coordinates
`devp`	product $[\mathbf{D}^e]\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}$
`diag`	diagonal of left hand side matrix
`diag_precon`	diagonal preconditioner vector
`disps`	global displacements and pore pressures
`eld`	element nodal displacements
`eload`	integrating point contribution to `bload`
`eps`	strain terms
`erate`	viscoplastic strain rate, $\left\{\dot{\boldsymbol{\epsilon}}^{vp}\right\}$
`evp`	viscoplastic strain rate increment, $\left\{\boldsymbol{\delta\epsilon}^{vp}\right\}$
`evpt`	holds running total of viscoplastic strains, $\left\{\boldsymbol{\Delta\epsilon}^{vp}\right\}$
`flow`	holds $\{\partial Q/\partial\boldsymbol{\sigma}\}$
`fun`	solid shape functions
`funf`	fluid shape functions
`gamma`	small local array

`gc`	integrating point coordinates
`gg`	small local array
`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`kay`	property matrix
`kd`	right-hand side element matrix from “Biot” analysis
`ke`	element “stiffness” matrix
`km`	element stiffness matrix
`kc`	element conductivity matrix
`kv`	global stiffness matrix
`lf`	input load/time function
`loads`	nodal velocities and pressures
`m1`	holds holds $\partial\sigma_m/\partial\boldsymbol{\sigma}$
`m2`	holds holds $\partial J_2/\partial\boldsymbol{\sigma}$
`m3`	holds holds $\partial J_3/\partial\boldsymbol{\sigma}$
`nd1`	product $[\texttt{fun}]^{\mathrm{T}}[\texttt{deriv(1,:)}]$
`nd2`	product $[\texttt{fun}]^{\mathrm{T}}[\texttt{deriv(2,:)}]$
`ndf1`	product $[\texttt{fun}]^{\mathrm{T}}[\texttt{derivf(1,:)}]$
`ndf2`	product $[\texttt{fun}]^{\mathrm{T}}[\texttt{derivf(2,:)}]$
`nfd1`	product $[\texttt{funf}]^{\mathrm{T}}[\texttt{deriv(1,:)}]$
`nfd2`	product $[\texttt{funf}]^{\mathrm{T}}[\texttt{deriv(2,:)}]$
`newdis`	“new” displacements and pore pressures
`oldis`	“old” displacements and pore pressures
`oldlds`	nodal velocities and pressures from previous iteration
`p`	“descent” vector used in (3.22)
`pb`	unsymmetric global band “stiffness” matrix
`phi0`	used in element-by-element “gather” algorithm
`phi1`	used in element-by-element “scatter” algorithm
`points`	integrating point local coordinates
`prop`	element properties
`r`	residual vector
`rt`	initial residual vector
`s`	small local vector
`sigma`	stress terms
`stress`	stress terms
`store`	“penalty” degrees of freedom
`storkd`	stores augmented diagonal terms
`storke`	element matrix storage
`store_kc`	stores element `kc` matrices
`tensor`	holds running total of all integrating point stress terms
`u`	gather/scatter array
`utemp`	gather/scatter array
`uvel`	element nodal x-velocity
`val`	nodal loads weighting factors
`value`	fixed vales of freedoms
`vol`	related to the volumetric strain

`volf`	used to compute coupling matrix
`vvel`	element nodal y-velocity
`weights`	weighting coefficients
`work`	working space
`x`	"old" solution vector
`xmul`	gather/scatter array
`xnew`	"new" solution vector
`x_coords`	x-coordinates of mesh layout
`y`	gather/scatter array
`y1`	gather/scatter array
`y_coords`	y-coordinates of mesh layout

9.2 Exercises

1. The mesh shown in Figure 9.17 is to be used to model 1D flow in the x-direction between two horizontal plates situated at $y = 0.0$ and $y = -3.0$. Velocity boundary conditions are that the top plate is moved with a velocity of $u = 3.0$ relative to the bottom plate which is fixed at $u = 0$. Pressure boundary conditions are that the pressure on the left and right vertical boundaries are set to $p = 1.0$ and $p = -1.0$ respectively, giving a pressure gradient of $\partial p/\partial x = -2.0$. Given that $\mu = \rho =$

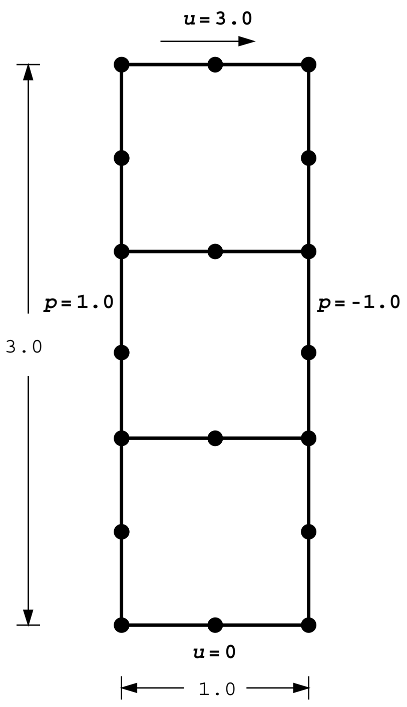


Figure 9.17

1.0 (`visc` and `rho` respectively in programming terminology), use Program 9.1 to estimate the steady state mid-plane velocity.
(Ans: $u = 3.75$)

2. Use Program 9.3 to reproduce the Mandel–Cryer effect (see e.g. Lambe and Whitman, 1969, Fig 27.10), in which the excess pore pressures beneath the center of a uniformly loaded flexible strip footing temporarily rise before they start to dissipate.

3. Use a trial and error approach with the example accompanying Program 9.4 in this chapter to estimate the deviator stress loading rate that gives a failure load exactly half way between the drained and undrained solutions.
(Ans: $\mathrm{d}D/\mathrm{d}t \approx 2.17$ gives $D_\mathrm{f} \approx 150$ kN/m^2)

References

Biot MA 1941 General theory of three-dimensional consolidation. *J Appl Phys* **12**, 155–164.

Griffiths DV 1985 The effect of pore fluid compressibility on failure loads in elasto-plastic soils. *Int J Numer Anal Methods Geomech* **9**, 253–259.

Griffiths DV 1994 Coupled analyses in geomechanics. In *Visco-Plastic Behavior of Geomaterials* (eds. Cristescu ND and Gioda G). Springer-Verlag, Wien, New York, pp. 245–317. Chapter 5.

Kidger DJ 1994 Visualisation of three-dimensional processes in geomechanics computations. In *Proceedings of the 8th International Conference on Computational Methods and Advances in Geomechanics* (eds. Siriwardane H and Zaman M). A. A. Balkema, Rotterdam, pp. 453–457.

Lambe T and Whitman R 1969 *Soil Mech.* John Wiley & Sons, Chichester, New York.

Schiffman RL 1960 Field applications of soil consolidation, time-dependent loading and variable permeability. Technical Report 248, Highway Research Board, Washington, D.C.

Smith IM and Hobbs R 1976 Biot analysis of consolidation beneath embankments. *Géotechnique* **26**, 149–171.

10

Eigenvalue Problems

10.1 Introduction

The ability to solve eigenvalue problems is important in many aspects of finite element work. For example, the number of zero eigenvalues of an element "stiffness" matrix (its rank deficiency) is an important guide to the suitability of that element. In that context, the problem to be solved is just:

$$[\mathbf{k}]\{\mathbf{u}\} = \lambda\{\mathbf{u}\} \tag{10.1}$$

which is the eigenvalue problem in "standard form" (3.83). More often, the eigenvalue equation will describe a physical situation such as free vibration of a solid or fluid. For example, (2.19) after assembly, for a freely vibrating elastic solid becomes,

$$[\mathbf{K}_m]\{\mathbf{X}\} = \omega^2[\mathbf{M}_m]\{\mathbf{X}\} \tag{10.2}$$

which can readily be converted to "standard form" by the procedure outlined in Section 3.9.1. In this case, the global mass matrix $[\mathbf{M}_m]$ may be "lumped" or "consistent" (Section 3.7.7).

The present Chapter describes four programs for the determination of eigenvalues and eigenvectors of such elastic structures and solids. Different algorithms and storage strategies are employed in the various cases. Since elastic solids are considered, the programs can be viewed as extensions of the programs described in Chapters 4 and 5. The same terminology is used. Program 10.1 computes the natural frequencies and mode shapes of strings of beam elements, and Program 10.2 does the same for planar elastic solids using 4- or 8-node quadrilaterals. Both programs use Jacobi's method. Programs 10.3 and 10.4 use the Lanczos algorithm (Section 3.9.2) to calculate natural frequencies and mode shapes of elastic solids. The first of these two programs uses a global assembly strategy while the second adopts an element-by-element approach looking forward to the parallelised example in Chapter 12.

Program 10.1 Eigenvalue analysis of elastic beams using 2-node beam elements. Lumped mass.

```
PROGRAM p101
!-------------------------------------------------------------------------
! Program 10.1 Eigenvalue analysis of elastic beams using 2-node
!              beam elements. Lumped mass.
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,idiag,iel,ifail,j,k,nband,ndof=4,nels,neq,nmodes,nn,nod=2,    &
   nodof=2,nprops=2,np_types,nr
 REAL(iwp)::d12=12.0_iwp,one=1.0_iwp,pt5=0.5_iwp,penalty=1.e20_iwp,       &
   etol=1.0e-30_iwp,zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),kdiag(:),nf(:,:),num(:)
 REAL(iwp),ALLOCATABLE::diag(:),ell(:),kh(:),km(:,:),ku(:,:),kv(:),       &
   mm(:,:),prop(:,:),rrmass(:),udiag(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,np_types; nn=nels+1
 ALLOCATE(nf(nodof,nn),km(ndof,ndof),num(nod),g(ndof),mm(ndof,ndof),      &
   ell(nels),etype(nels),g_g(ndof,nels),prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(diag(0:neq),udiag(0:neq),kdiag(neq),rrmass(0:neq))
!-----------------------loop the elements to find global array sizes------
 nband=0; kdiag=0
 elements_1: DO iel=1,nels
   num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   IF(nband<bandwidth(g))nband=bandwidth(g); CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 WRITE(11,'(A,I5,A,/,A,I5,/,A,I5)')" There are",neq," equations",         &
   " The half-bandwidth (including diagonal) is",nband+1,                 &
   " The skyline storage is",kdiag(neq)
!-----------------------global stiffness and mass matrix assembly---------
 ALLOCATE(ku(neq,nband+1),kv(kdiag(neq)),kh(kdiag(neq)))
 diag=zero; ku=zero
 elements_2: DO iel=1,nels
   g=g_g(:,iel); mm=zero; mm(1,1)=pt5*prop(2,etype(iel))*ell(iel)
   mm(3,3)=mm(1,1); mm(2,2)=mm(1,1)*ell(iel)**2/d12; mm(4,4)=mm(2,2)
   CALL formlump(diag,mm,g); CALL beam_km(km,prop(1,etype(iel)),ell(iel))
   CALL formku(ku,km,g)
 END DO elements_2
!-----------------------reduce to standard eigenvalue problem-------------
 rrmass(1:)=one/SQRT(diag(1:))
 DO i=1,neq; IF(i<=neq-nband)THEN; k=nband+1; ELSE; k=neq-i+1; END IF
   DO j=1,k; ku(i,j)=ku(i,j)*rrmass(i)*rrmass(i+j-1); END DO
 END DO
!-----------------------convert to skyline form---------------------------
 kh(1)=ku(1,1); k=1
 DO i=2,neq; idiag=kdiag(i)-kdiag(i-1)
   DO j=1,idiag; k=k+1; kh(k)=ku(i+j-idiag,1-j+idiag); END DO
 END DO
!-----------------------extract the eigenvalues---------------------------
 CALL bandred(ku,diag,udiag); ifail=1; CALL bisect(diag,udiag,etol,ifail)
 WRITE(11,'(/A)')" The eigenvalues are:"; WRITE(11,'(6E12.4)')diag(1:)
```

```
!----------------------extract the eigenvectors-------------------------
 READ(10,*)nmodes
 DO i=1,nmodes
   kv=kh;kv(kdiag)=kv(kdiag)-diag(i); kv(1)=kv(1)+penalty
   udiag=zero; udiag(1)=kv(1)
   CALL sparin_gauss(kv,kdiag); CALL spabac_gauss(kv,udiag,kdiag)
   udiag=rrmass*udiag; WRITE(11,'(A,I3,A)')" Eigenvector number",i," is:"
   WRITE(11,'(6E12.4)')udiag(1:)/MAXVAL(ABS(udiag(1:)))
 END DO
STOP
END PROGRAM p101
```

Scalar integers:

`i`	simple counter
`iel`	simple counter
`idiag`	skyline bandwidth
`iwp`	`SELECTED_REAL_KIND(15)`
`ifail`	warning flag from `bisect` subroutine
`j`	simple counters
`k`	simple counters
`nband`	bandwidth of upper triangle
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nmodes`	number of eigenvectors required
`nn`	number of nodes in the mesh
`nod`	number of nodes per elements
`nodof`	number of degrees of freedom per node
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes

Scalar reals:

`d12`	set to 12.0
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`pt5`	set to 0.5
`etol`	eigenvalue tolerance set to 1×10^{-30}
`zero`	set to 0.0

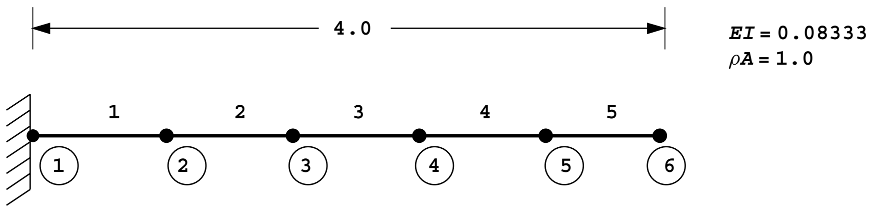
Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`kdiag`	diagonal term location vector
`nf`	nodal freedom matrix
`num`	element node number vector

Dynamic real arrays:

`diag`	global lumped mass vector
`ell`	element lengths vector
`kh`	element stiffness matrix
`km`	element stiffness matrix
`ku`	global stiffness matrix
`kv`	global stiffness matrix
`mm`	element lumped mass matrix
`prop`	element properties matrix
`rrmass`	reciprocal square rooted global lumped mass matrix
`udiag`	working space vector

This program illustrates a natural frequency analysis of a typical "string" of beam elements, and can be thought of as an extension to Program 4.3. The natural frequencies of the simple cantilever shown in Figure 10.1 are to be found. The first line of data gives the number of elements `nels` which equals 5 in this case. The next line gives the number of properties `np_types` which for a uniform beam equals 1. The two properties are then read in as the flexural stiffness *EI* read as 0.08333 and the mass per unit length ρA read



```
nels
5

np_types
1

prop(ei,rhoa)
0.08333  1.0

etype(not needed)

ell
0.8  0.8  0.8  0.8  0.8

nr,(k,nf(:,k),i=1,nr)
1
1 0 0

nmodes
3
```

Figure 10.1 Mesh and data for Program 10.1 example

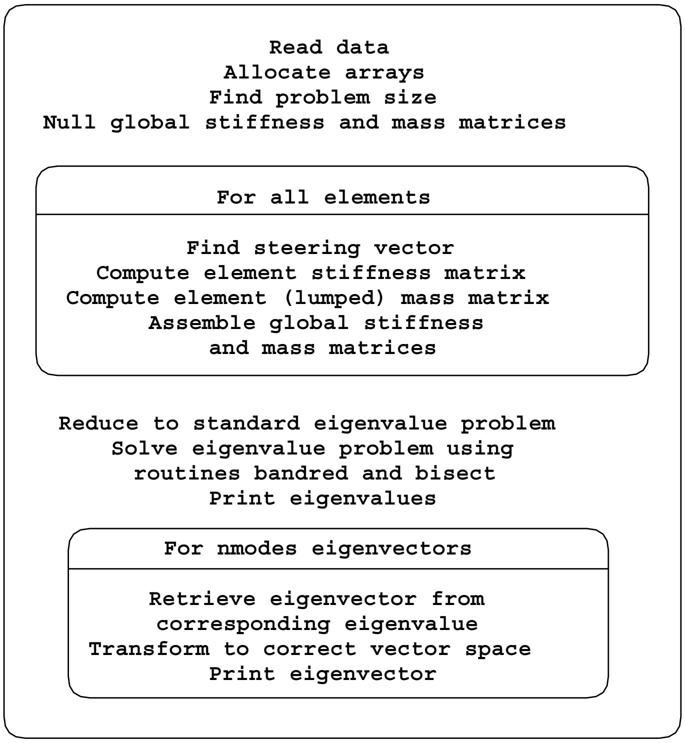
as 1.0. With only one property type, the `etype` data is not needed. The next line reads the lengths of the elements, which in this case are all equal to 0.8. The nodal freedom data then follows by fixing the cantilever end to have no translation or rotation. The final line of data reads `nmodes` which represents the number of eigenvectors (or mode shapes) to be computed. In this example the first three are requested.

After the usual preliminary steps to establish the global array size, the elements are assembled into the global stiffness and mass matrices. In this case, the global stiffness $[\mathbf{K}_m]$ is stored in `ku` as an upper band rectangle by library subroutine `formku` (see Figure 3.18), and the global lumped mass matrix $[\mathbf{M}_m]$ is stored as a vector in `diag` by library subroutine `formlump`. The structure of the program is shown in Figure 10.2.

The diagonal element mass matrices $[\mathbf{m}_m]$ are held in `mm` and use the following lumping (see e.g. Cook *et al.*, 1989):

$$[\mathbf{m}_m] = \frac{\rho AL}{2}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & L^2/12 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & L^2/12 \end{bmatrix} \tag{10.3}$$

By factorising `diag` and altering the appropriate terms in `ku` (see Section 3.9.1), the symmetrical matrix for the standard eigenvalue problem is retrieved (still called `ku`). The



```
Read data
Allocate arrays
Find problem size
Null global stiffness and mass matrices
    For all elements
        Find steering vector
        Compute element stiffness matrix
        Compute element (lumped) mass matrix
        Assemble global stiffness
        and mass matrices
Reduce to standard eigenvalue problem
Solve eigenvalue problem using
routines bandred and bisect
Print eigenvalues
    For nmodes eigenvectors
        Retrieve eigenvector from
        corresponding eigenvalue
        Transform to correct vector space
        Print eigenvector
```

Figure 10.2 Structure chart for Program 10.1

```
           There are   10 equations, the half-bandwidth is    3
                                and the skyline storage is   31

                           The eigenvalues are:
    0.3823E-02  0.1278E+00  0.8511E+00  0.2765E+01  0.6323E+01  0.1147E+02
                0.1748E+02  0.2324E+02  0.2756E+02  0.3225E+02
                          Eigenvector number  1 is:
    0.6303E-01  0.1499E+00  0.2275E+00  0.2539E+00  0.4579E+00  0.3154E+00
                0.7229E+00  0.3423E+00  0.1000E+01  0.3485E+00
                          Eigenvector number  2 is:
    0.2307E+00  0.4535E+00  0.5431E+00  0.2347E+00  0.5094E+00 -0.3411E+00
                0.2270E-01 -0.8308E+00 -0.7287E+00 -0.1000E+01
                          Eigenvector number  3 is:
    0.2888E+00  0.4155E+00  0.3179E+00 -0.3976E+00 -0.1579E+00 -0.5696E+00
               -0.2684E+00  0.3779E+00  0.3425E+00  0.1000E+01
```

Figure 10.3 Results from Program 10.1 example

eigenvalues of this band matrix (ω^2) are then calculated using Jacobi's method, which employs library subroutines `bandred` and `bisect`.

The first `nmodes` eigenvectors are then extracted by finding the relevant non-trivial solutions to the homogeneous equations and transforming them back to the correct vector space. In order to do this the global matrix is converted to skyline vector storage form (`kv`) and the eigenvectors solved for using subroutines `sparin_gauss` and `spabac_gauss`. In this example, the first three eigenvectors are computed and normalised to a vector length (Euclidean norm) of unity. The results are all shown in Figure 10.3.

As a check, the computed results indicate a fundamental frequency $\omega = \sqrt{0.0038} = 0.062$ which should be compared with the analytical result (e.g. Chopra, 1995) for a slender beam of,

$$\omega = \frac{3.516}{L^2}\sqrt{\frac{EI}{\rho A}} = 0.063 \tag{10.4}$$

Program 10.2 Eigenvalue analysis of an elastic solid in plane strain using 4- or 8-node rectangular quadrilaterals. Lumped mass. Mesh numbered in *x*- or *y*-direction.

```
PROGRAM p102
!-------------------------------------------------------------------------
! Program 10.2 Eigenvalue analysis of an elastic solid in plane strain
!              using 4- or 8-node rectangular quadrilaterals. Lumped mass.
!              Mesh numbered in x- or y-direction.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,idiag,iel,ifail,j,k,nband,ndim=2,ndof,nels,neq,nmodes,nn,nod, &
   nodof=2,nprops=3,np_types,nr,nxe,nye
 REAL(iwp)::area,etol=1.e-30_iwp,one=1.0_iwp,penalty=1.e20_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays-------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g (:,:),g_num(:,:),kdiag(:),nf(:,:),&
   num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),diag(:),g_coord(:,:),kh(:),km(:,:),    &
   ku(:,:),kv(:),mm(:,:),prop(:,:),rrmass(:),udiag(:),x_coords(:),        &
   y_coords(:)
!-----------------------input and initialisation--------------------------
```

```
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,nod,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),g_coord(ndim,nn),coord(nod,ndim),mm(ndof,ndof),     &
   g_num(nod,nels),num(nod),km(ndof,ndof),g(ndof),g_g(ndof,nels),          &
   prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(diag(0:neq),udiag(0:neq),kdiag(neq),rrmass(0:neq))
!-----------------------loop the elements to find global array sizes------
 nband=0; kdiag=0
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
   IF(nband<bandwidth(g))nband=bandwidth(g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 WRITE(11,'(A,I5,A,/,A,I5,/,A,I5)')" There are",neq," equations",          &
   " The half-bandwidth (including diagonal) is",nband+1,                  &
   " The skyline storage is",kdiag(neq)
!-----------------------global stiffness and mass matrix assembly---------
 ALLOCATE(ku(neq,nband+1),kv(kdiag(neq)),kh(kdiag(neq)))
 diag=zero; ku=zero
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
   CALL rect_km(km,coord,prop(1,etype(iel)),prop(2,etype(iel)))
   CALL formku(ku,km,g)
   area=(MAXVAL(coord(:,1))-MINVAL(coord(:,1)))*                           &
        (MAXVAL(coord(:,2))-MINVAL(coord(:,2)))
   CALL elmat(area,prop(3,etype(iel)),mm); CALL formlump(diag,mm,g)
 END DO elements_2
!-----------------------reduce to standard eigenvalue problem-------------
 rrmass(1:)=one/SQRT(diag(1:))
 DO i=1,neq; IF(i<=neq-nband)THEN; k=nband+1; ELSE; k=neq-i+1; END IF
   DO j=1,k; ku(i,j)=ku(i,j)*rrmass(i)*rrmass(i+j-1); END DO
 END DO
!-----------------------convert to skyline form---------------------------
 kh(1)=ku(1,1); k=1
 DO i=2,neq; idiag=kdiag(i)-kdiag(i-1)
   DO j=1,idiag; k=k+1; kh(k)=ku(i+j-idiag,1-j+idiag); END DO
 END DO
!-----------------------extract the eigenvalues---------------------------
 CALL bandred(ku,diag,udiag); ifail=1; CALL bisect(diag,udiag,etol,ifail)
 WRITE(11,'(/A)')" The eigenvalues are:"; WRITE(11,'(6E12.4)')diag(1:)
!-----------------------extract the eigenvectors--------------------------
 READ(10,*)nmodes
 DO i=1,nmodes
   kv=kh;kv(kdiag)=kv(kdiag)-diag(i); kv(1)=kv(1)+penalty
   udiag=zero; udiag(1)=kv(1)
   CALL sparin_gauss(kv,kdiag); CALL spabac_gauss(kv,udiag,kdiag)
   udiag=rrmass*udiag; WRITE(11,'(A,I3,A)')" Eigenvector number",i," is:"
   WRITE(11,'(6E12.4)')udiag(1:)/MAXVAL(ABS(udiag(1:)))
   IF(i==1)CALL dismsh(udiag,nf,0.1_iwp,g_coord,g_num,13)
 END DO
STOP
END PROGRAM p102
```

New scalar integers:

`ndim`	number of dimensions
`nxe`	number of elements in the x-direction
`nye`	number of elements in the y-direction

New scalar reals:

`area`	element area

Scalar character:

`element`	element type

New dynamic integer arrays:

`g_num`	global element node numbers matrix

New dynamic real arrays:

`coord`	element nodal coordinates
`g_coord`	nodal coordinates for all elements
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout

This program is an extension of Program 5.1 for the analysis of elastic solids in plane strain. The element coordinates and steering information are produced by the geometry subroutine `geom_rect` otherwise the structure of the program has much in common with Program 10.1 (see Figure 10.2).

The example problem shown in Figure 10.4 is nominally the same as the beam analysed in Figure 10.1, namely an elastic solid cantilever 4.0 units long in the x-direction with a flexural rigidity of 0.08333. The solid modelled by the five 4-node elements in Figure 10.4 is two-dimensional however, so Poisson's ratio has been set to zero to remove the stiffening effect of plane strain. The mass density is set to unity.

Rather than performing the usual numerical integration loops, this program introduces the new subroutine `rect_km` which computes the stiffness matrix (`km`) of an elastic 4- or 8-node *rectangular* element in "closed form" based on `nip=4`. Subroutine `elmat` forms the lumped mass matrix (`mm`) for a 4- or 8-node quadrilateral based on its area. For a 4-node element, the lumped mass matrix `mm` is readily formed with eight diagonal

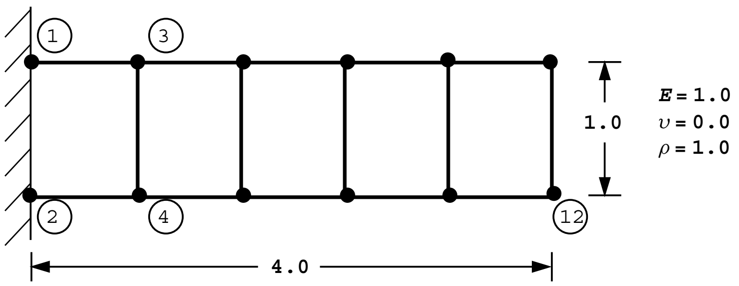


Figure 10.4 Mesh and data for first Program 10.2 example (*Continued on page 449*)

```
nxe  nye  nod
5     1    4

np_types
1

prop(e,v,rho)
1.0  0.0  1.0

etype (not needed)

x_coords, y_coords
0.0  0.8  1.6  2.4  3.2  4.0
0.0 -1.0

nr,(k,nf(:,k),i=1,nr)
2
1 0 0  2 0 0

nmodes
5
```

Figure 10.4 (*Continued from page 448*)

terms, in which one quarter of the total mass of the element is lumped at each node in each direction. The element stiffness and mass matrices are assembled into their global counterparts ku and diag as discussed previously, and the remainder of the program is identical to Program 10.1.

The first line of data provides the number of elements in the x- and y-directions (nxe and nye) and the number of nodes per element (nod). Three properties are required in a problem such as this, namely, Young's modulus E, Poisson's ratio ν and the mass density ρ. This is followed by the mesh coordinate data (x_coords and y_coords) and the boundary condition data, which involves fully fixing the nodes at the built-in end of the cantilever. The final line of data as before reads nmodes, which represents the number of eigenvectors (or mode shapes) to be computed. In this example, five eigenvectors are requested.

From the results shown in Figure 10.5, it can be seen that the fundamental frequency, printed as $\omega = \sqrt{0.004595} = 0.068$, is rather higher than the value of 0.062 calculated by Program 10.1 for a slender beam. Thus, the 2D solid, represented by 4-node elements with full integration, is a poor representation of a slender beam, at least in the flexural modes. The elements are too "stiff". The longitudinal modes as indicated by the third eigenvalue and eigenvector are more accurately modelled by this element however; the computed frequency, printed as $\omega = \sqrt{0.1529} = 0.391$ is in good agreement with the analytical solution of $\omega = \pi/2L\sqrt{E/\rho} = 0.393$.

A much better representation of flexural modes of "beams" made up of solid elements is achieved by the use of 8-node quadrilaterals, so the second example uses this superior element to solve the same problem, however the process of mass lumping is not obvious in this case. For example, the summation of rows of the consistent matrix leads to negative values at the corners. It can be shown however, (Smith, 1977) that a reasonable approximation is to lump the mass to the mid-point and corner nodes in the ratio 4:1, thus 1/20 of the total mass is assigned to each corner node and 1/5 to each mid-side node in each direction. This weighting is assigned to the 8-node element by subroutine `elmat`.

The mesh and data shown in Figure 10.6 are virtually identical to those used for the analysis with 4-node elements. The only differences lies in the data for `nod`, which is now read as 8, and the boundary condition data at the built-in end of the cantilever involves 3 fixed nodes.

```
There are   20 equations, the half-bandwidth is    7
                      and the skyline storage is  114

The eigenvalues are:
 0.4595E-02  0.1053E+00  0.1529E+00  0.5169E+00  0.1226E+01  0.1288E+01
 0.2058E+01  0.2352E+01  0.2415E+01  0.2710E+01  0.2719E+01  0.3114E+01
 0.3125E+01  0.3188E+01  0.3381E+01  0.3665E+01  0.3960E+01  0.4211E+01
 0.4962E+01  0.6097E+01
Eigenvector number  1 is:
 0.7195E-01 -0.6932E-01 -0.7195E-01 -0.6932E-01  0.1224E+00 -0.2362E+00
-0.1224E+00 -0.2362E+00  0.1526E+00 -0.4662E+00 -0.1526E+00 -0.4662E+00
 0.1662E+00 -0.7285E+00 -0.1662E+00 -0.7285E+00  0.1695E+00 -0.1000E+01
-0.1695E+00 -0.1000E+01
Eigenvector number  2 is:
 0.2487E+00 -0.4240E+00 -0.2487E+00 -0.4240E+00  0.1022E+00 -0.8727E+00
-0.1022E+00 -0.8727E+00 -0.2584E+00 -0.7980E+00  0.2584E+00 -0.7980E+00
-0.5732E+00 -0.7556E-01  0.5732E+00 -0.7556E-01 -0.6870E+00  0.1000E+01
 0.6870E+00  0.1000E+01
Eigenvector number  3 is:
 0.3090E+00  0.1555E-14  0.3090E+00  0.1572E-14  0.5878E+00  0.3059E-14
 0.5878E+00  0.2986E-14  0.8090E+00  0.2674E-14  0.8090E+00  0.2728E-14
 0.9511E+00 -0.5347E-16  0.9511E+00  0.0000E+00  0.1000E+01 -0.3174E-14
 0.1000E+01 -0.3478E-14
Eigenvector number  4 is:
 0.2413E+00 -0.9151E+00 -0.2413E+00 -0.9151E+00 -0.4612E+00 -0.8557E+00
 0.4612E+00 -0.8557E+00 -0.5141E+00  0.3742E+00  0.5141E+00  0.3742E+00
 0.4120E+00  0.6579E+00 -0.4120E+00  0.6579E+00  0.1000E+01 -0.7050E+00
-0.1000E+01 -0.7050E+00
Eigenvector number  5 is:
 0.2575E+00  0.9693E+00 -0.2575E+00  0.9693E+00  0.8105E+00 -0.2304E+00
-0.8105E+00 -0.2304E+00 -0.3806E+00 -0.5583E+00  0.3806E+00 -0.5583E+00
-0.6787E-01  0.6923E+00  0.6787E-01  0.6923E+00  0.1000E+01 -0.2476E+00
-0.1000E+01 -0.2476E+00
```

Figure 10.5 Results from first Program 10.2 example

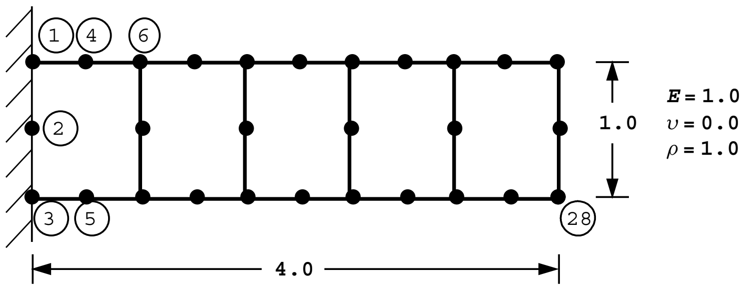


Figure 10.6 Mesh and data for second Program 10.2 example (*Continued on page 451*)

```
nxe  nye  nod
5    1    8

np_types
1

prop(e,v,rho)
1.0  0.0  1.0

etype (not needed)

x_coords, y_coords
0.0  0.8  1.6  2.4  3.2  4.0
0.0 -1.0

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

nmodes
5
```

Figure 10.6 (*Continued from page 450*)

```
There are   50 equations, the half-bandwidth is   15
                 and the skyline storage is  515

The eigenvalues are:
 0.3641E-02  0.9363E-01  0.1532E+00  0.4932E+00  0.1255E+01  0.1270E+01
 0.1524E+01  0.2390E+01  0.3006E+01  0.3321E+01  0.3803E+01  0.4927E+01
 0.5010E+01  0.5648E+01  0.5844E+01  0.6508E+01  0.6935E+01  0.7054E+01
 0.7097E+01  0.7420E+01  0.9235E+01  0.9569E+01  0.9828E+01  0.9873E+01
 0.1050E+02  0.1056E+02  0.1079E+02  0.1082E+02  0.1101E+02  0.1119E+02
 0.1124E+02  0.1152E+02  0.1172E+02  0.1178E+02  0.1388E+02  0.1582E+02
 0.1634E+02  0.1793E+02  0.1915E+02  0.1998E+02  0.2072E+02  0.2263E+02
 0.2639E+02  0.3313E+02  0.3879E+02  0.4823E+02  0.5123E+02  0.5894E+02
 0.6294E+02  0.7198E+02
Eigenvector number  1 is:
 0.3907E-01 -0.2065E-01 -0.3907E-01 -0.2065E-01  0.7253E-01 -0.7030E-01
-0.2551E-13 -0.7029E-01 -0.7253E-01 -0.7030E-01  0.1004E+00 -0.1444E+00
-0.1004E+00 -0.1444E+00  0.1228E+00 -0.2385E+00 -0.3805E-13 -0.2384E+00
-0.1228E+00 -0.2385E+00  0.1401E+00 -0.3480E+00 -0.1401E+00 -0.3480E+00
 0.1524E+00 -0.4691E+00 -0.3349E-13 -0.4690E+00 -0.1524E+00 -0.4691E+00
 0.1606E+00 -0.5976E+00 -0.1606E+00 -0.5976E+00  0.1652E+00 -0.7307E+00
-0.3567E-13 -0.7306E+00 -0.1652E+00 -0.7307E+00  0.1672E+00 -0.8651E+00
-0.1672E+00 -0.8651E+00  0.1677E+00 -0.1000E+01 -0.3424E-13 -0.9998E+00
-0.1677E+00 -0.1000E+01
.
.
.

Eigenvector number  5 is:
 0.1827E+00  0.6303E-02  0.1827E+00 -0.6303E-02  0.3434E+00  0.1255E-01
 0.3188E+00 -0.1552E-12  0.3434E+00 -0.1255E-01  0.4397E+00  0.2189E-01
 0.4397E+00 -0.2189E-01  0.4831E+00 -0.1363E-01  0.3530E+00  0.7292E-13
 0.4831E+00  0.1363E-01  0.2961E+00 -0.5326E-01  0.2961E+00  0.5326E-01
 0.7352E-01  0.1180E-01  0.1981E+00  0.8243E-13  0.7352E-01 -0.1180E-01
 0.2596E-01  0.8228E-01  0.2596E-01 -0.8228E-01 -0.2472E-01 -0.3080E-01
-0.3033E+00 -0.1238E-12 -0.2472E-01  0.3080E-01 -0.5452E+00 -0.1962E+00
-0.5452E+00  0.1962E+00 -0.1000E+01  0.5836E+00 -0.8896E-01  0.1023E-12
-0.1000E+01 -0.5836E+00
```

Figure 10.7 Results from second Program 10.2 example

The output shown in Figure 10.7 indicates a fundamental frequency of $\omega = \sqrt{0.003641} = 0.060$, which is in closer agreement with the value of 0.062 produced by Program 10.1. The program outputs the fundamental mode shape to the graphics output file `fe95.dis` and this is shown in Figure 10.8.

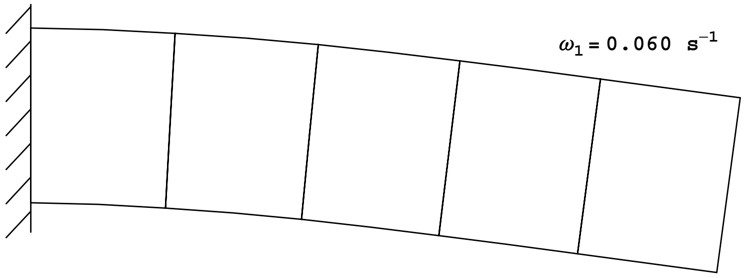


Figure 10.8 Fundamental mode shape from second Program 10.2 example

Program 10.3 Eigenvalue analysis of an elastic solid in plane strain using 4-node rectangular quadrilaterals. Lanczos Method. Consistent mass. Mesh numbered in x- or y-direction.

```
PROGRAM p103
!-------------------------------------------------------------------------
! Program 10.3: Eigenvalue analysis of an elastic solid in plane strain
!               using 4-node rectangular quadrilaterals. Lanczos Method.
!               Consistent mass. Mesh numbered in x- or y-direction.
!-------------------------------------------------------------------------
 USE main;  USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,iflag=-1,iters,jflag,k,lalfa,leig,lp=6,lx,lz,nband=0,     &
   ndim=2,ndof,neig=0,nels,neq,nip=4,nmodes,nn,nod,nodof=2,nprops=3,      &
   np_types,nr,nst=3,nxe,nye
 REAL(iwp)::acc,det,el,er,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!----------------------------- dynamic arrays----------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),jeig(:,:),nf(:,:),&
   nu(:),num(:)
 REAL(iwp),ALLOCATABLE::alfa(:),bee(:,:),beta(:),coord(:,:),dee(:,:),     &
   del(:),der(:,:),deriv(:,:),diag(:),ecm(:,:),eig(:),fun(:),g_coord(:,:),&
   jac(:,:),kb(:,:),km(:,:),mb(:,:),mm(:,:),points(:,:),prop(:,:),ua(:),  &
   udiag(:),va(:),v_store(:,:),weights(:),w1(:),x(:),x_coords(:),y(:,:),  &
   y_coords(:),z(:,:)
!--------------------------input and initialisation----------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,nod,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),g_coord(ndim,nn),    &
   coord(nod,ndim),fun(nod),jac(ndim,ndim),weights(nip),g_num(nod,nels),  &
   der(ndim,nod),deriv(ndim,nod),bee(nst,ndof),num(nod),km(ndof,ndof),    &
   g(ndof),g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),                   &
   prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 READ(10,*)nmodes,el,er,lalfa,laig,lx,lz,acc
 ALLOCATE(eig(leig),x(lx),del(lx),nu(lx),jeig(2,leig),alfa(lalfa),        &
   beta(lalfa),z(lz,leig))
!-------- loop the elements to find nband and set up global arrays -------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
   IF(nband<bandwidth(g))nband=bandwidth(g)
 END DO elements_1
 WRITE(11,'(A,I5,A,/,A,I5,/,A,I5)')" There are",neq," equations",          &
   " The half-bandwidth (including diagonal) is",nband+1
 ALLOCATE(kb(neq,nband+1),mb(neq,nband+1),ua(0:neq),va(0:neq),             &
   diag(0:neq),udiag(0:neq),w1(0:neq),y(0:neq,leig),v_store(0:neq,lalfa))
 kb=zero; mb=zero; ua=zero; va=zero; eig=zero; jeig=0; x=zero; del=zero
 nu=0; alfa=zero; beta=zero; diag=zero; udiag=zero; w1=zero; y=zero; z=zero
 CALL sample(element,points,weights)
!----------------- element stiffness integration and assembly-------------
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
   km=zero; mm=zero
   integrating_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); call beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
     call ecmat(ecm,fun,ndof,nodof)
     mm=mm+ecm*det*weights(i)*prop(3,etype(iel))
   END DO integrating_pts_1
   CALL formkb(kb,km,g); CALL formkb(mb,mm,g)
 END DO elements_2; CALL cholin(mb)
!-----------------------------find eigenvalues--------------------------
 DO iters=1,lalfa
   call lancz1(neq,el,er,acc,leig,lx,lalfa,lp,iflag,ua,va,eig,jeig,neig,x,&
   del,nu,alfa,beta,v_store)
   IF(iflag==0)EXIT
   IF(iflag>1)THEN; WRITE(11,'(A,I5)')                                     &
       " Lancz1 is signalling failure, with iflag = ",   iflag; STOP
   END IF
!--- iflag = 1 therefore form u + a * v  ( candidate for ebe ) -----------
   udiag=va; CALL chobk2(mb,udiag); CALL banmul(kb,udiag,diag)
   CALL chobk1(mb,diag); ua=ua+diag
 END DO
!--- iflag = 0 therefore write out the spectrum  -------------------------
 WRITE(11,'(A,I5,A/)')" It took",iters," iterations"
 WRITE(11,'(3(A,E12.4))')" Eigenvalues in the range",el," and",er," are:"
 WRITE(11,'(6E12.4)')eig(1:neig)
!------------------- calculate the eigenvectors -------------------------
 IF(neig>10)neig=10
  call lancz2(neq,lalfa,lp,eig,jeig,neig,alfa,beta,lz,jflag,y,w1,z,v_store)
!------------------if jflag is zero  calculate the eigenvectors ---------
 IF(jflag==0)THEN
   DO i=1,nmodes
     udiag(:)=y(:,i); CALL chobk2(mb,udiag)
     WRITE(11,'(" Eigenvector number",I4," is:")')i
     WRITE(11,'(6E12.4)')udiag(1:)/MAXVAL(ABS(udiag(1:)))
   END DO
 ELSE
! lancz2 fails
   WRITE(11,'(A,I5)')" Lancz2 is signalling failure with jflag = ",jflag
 END IF
STOP
END PROGRAM p103
```

New scalar integers:

`iflag`	no start vector specified, set to -1
`iters`	number of Lanczos iterations
`jflag`	equals zero if eigenvectors computed properly
`lalfa`	Lanczos iteration ceiling
`leig`	maximum number of eigenvalues in range
`lp`	unit number for diagnostic messages
`lx`	set to at least `3*leig`
`lz`	holds first dimension of array `z`
`neig`	holds the number of eigenvalues in `eig`
`nst`	number of stress (strain) terms

New scalar reals:

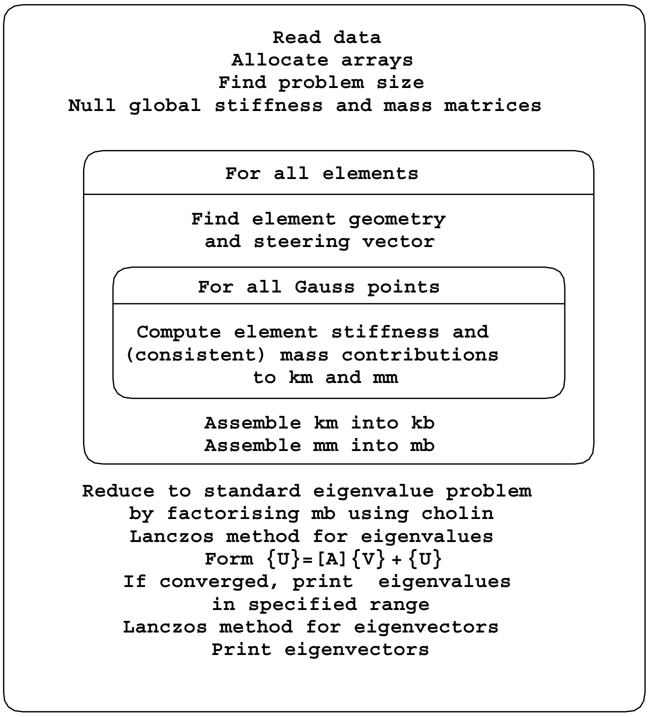
`acc`	convergence tolerance relative to largest eigenvalue
`det`	determinant of the Jacobian matrix
`el`	lower limit of eigenvalue spectrum
`er`	upper limit of eigenvalue spectrum

New dynamic integer arrays:

`jeig`	used to get the eigenvectors
`nu`	working array

New dynamic real arrays:

`alfa`	working array holding Lanczos tridiagonal matrix
`bee`	strain-displacement matrix
`beta`	working array holding Lanczos tridiagonal matrix
`dee`	stress–strain matrix
`del`	working array
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`ecm`	Gauss point contribution to consistent mass matrix
`eig`	holds the computed eigenvalues in increasing order
`fun`	shape functions
`jac`	Jacobian matrix
`kb`	global stiffness matrix
`mb`	global mass matrix
`points`	integrating point local coordinates
`ua`	working space vector
`udiag`	eigenvector
`va`	working space vector
`v_store`	holds the Lanczos vectors
`weights`	weighting coefficients
`w1`	working vector
`x`	working space vector
`y`	working space vector
`z`	working space vector



```
Read data
Allocate arrays
Find problem size
Null global stiffness and mass matrices
For all elements
    Find element geometry
    and steering vector
    For all Gauss points
        Compute element stiffness and
        (consistent) mass contributions
        to km and mm
    Assemble km into kb
    Assemble mm into mb
Reduce to standard eigenvalue problem
by factorising mb using cholin
Lanczos method for eigenvalues
Form {U}=[A]{V}+{U}
If converged, print eigenvalues
in specified range
Lanczos method for eigenvectors
Print eigenvectors
```

Figure 10.9 Structure chart for Program 10.3

In Programs 10.1 and 10.2 Jacobi transformation was used to solve the eigenvalue problem. Although reliable and robust this method is time consuming for large problems. For such problems, iterative methods are more attractive, for example the Lanczos method (Parlett and Reid, 1981). The process is described in Chapter 3, Section 3.9.2, and is used in the present program, whose structure chart is given as Figure 10.9. It should be noted that for the first time in this chapter, consistent mass has been used. The global mass matrix is therefore symmetric and banded and stored in array `mb` in the same way that the global stiffness matrix is stored in array `kb`. Assembly is achieved using the subroutine `formkb` (see Table 3.7 and Figure 3.18).

The principal new vectors are {**U**} and {**V**} (called `ua` and `va` in the program) as described in the structure chart of Figure 10.9. As seen in the above list, many of the arrays are used for working space. Library subroutines `lancz1` and `lancz2`[1] are used to retrieve the eigenvalues and eigenvectors respectively (see also Smith and Heshmati, 1983; Smith 1984).

The same 4-node problem considered previously is used to demonstrate this program as shown in Figure 10.10. Additional data read involves `el` and `er` which define the range of the eigenvalue search, and `lalfa`, `leig`, `lx`, `lz` and `acc` which are parameters related to the Lanczos algorithm (see notation section above).

[1]These names are aliases for `ea25a/ad` and `ea25e/ed` of HSL (2002).

```
nxe  nye  nod
5     1    4

np_types
1

prop(e,v,rho)
1.0  0.0  1.0

etype (not needed)

x_coords, y_coords
0.0  0.8  1.6  2.4  3.2  4.0
0.0 -1.0

nr,(k,nf(:,k),i=1,nr)
2
1 0 0  2 0 0

nmodes
5

el    er  lalfa  leig  lx   lz   acc
0.0  5.0   500    20   80  500  1.0e-6
```

Figure 10.10 Data for Program 10.3 and 10.4 examples

```
There are   20 equations and the half-bandwidth is    7
It took   22 iterations

Eigenvalues in the range  0.0000E+00 and  0.5000E+01 are:
 0.4931E-02  0.1499E+00  0.1555E+00  0.9573E+00  0.1493E+01  0.3043E+01
 0.4688E+01
Eigenvector number   1 is:
-0.7275E-01  0.7082E-01  0.7275E-01  0.7082E-01 -0.1229E+00  0.2394E+00
 0.1229E+00  0.2394E+00 -0.1522E+00  0.4700E+00  0.1522E+00  0.4700E+00
-0.1648E+00  0.7311E+00  0.1648E+00  0.7311E+00 -0.1677E+00  0.1000E+01
 0.1677E+00  0.1000E+01
Eigenvector number   2 is:
-0.2156E+00  0.4086E+00  0.2156E+00  0.4086E+00 -0.6107E-01  0.7885E+00
 0.6107E-01  0.7885E+00  0.2617E+00  0.6519E+00 -0.2617E+00  0.6519E+00
 0.5123E+00 -0.5054E-01 -0.5123E+00 -0.5054E-01  0.5925E+00 -0.1000E+01
-0.5925E+00 -0.1000E+01
Eigenvector number   3 is:
 0.3090E+00 -0.2022E-06  0.3090E+00 -0.2015E-06  0.5878E+00 -0.3972E-06
 0.5878E+00 -0.3961E-06  0.8090E+00 -0.3516E-06  0.8090E+00 -0.3508E-06
 0.9511E+00 -0.3366E-07  0.9511E+00 -0.3273E-07  0.1000E+01  0.4045E-06
 0.1000E+01  0.4053E-06
Eigenvector number   4 is:
-0.1193E+00  0.9255E+00  0.1193E+00  0.9255E+00  0.4733E+00  0.5949E+00
-0.4733E+00  0.5949E+00  0.3215E+00 -0.6339E+00 -0.3215E+00 -0.6339E+00
-0.5413E+00 -0.5437E+00  0.5413E+00 -0.5437E+00 -0.1000E+01  0.9777E+00
 0.1000E+01  0.9777E+00
Eigenvector number   5 is:
-0.8090E+00 -0.4830E-07 -0.8090E+00 -0.5335E-07 -0.9511E+00 -0.4265E-07
-0.9511E+00 -0.4216E-07 -0.3090E+00  0.3175E-07 -0.3090E+00  0.3655E-07
 0.5878E+00  0.3762E-07  0.5878E+00  0.4084E-07  0.1000E+01 -0.7482E-07
 0.1000E+01 -0.7508E-07
```

Figure 10.11 Results from Program 10.3 example

The output from the program is listed as Figure 10.11. It consists of the number of Lanczos iterations (22 in this case) and the computed eigenvalues in the range $0.0 < \omega^2 < 5.0$. The first five eigenvectors are also printed.

Since the consistent mass assumption was made, the eigenvalues differ somewhat from those computed by Program 10.2. For example the first eigenvalue is computed as $\omega =$

$\sqrt{0.004931} = 0.070$ as compared with 0.068 previously computed for a similar problem with lumped mass.

Program 10.4 Eigenvalue analysis of an elastic solid in plane strain using 4-node rectangular quadrilaterals. Lanczos Method. Lumped mass. Element-by-element formulation. Mesh numbered in x- or y-direction.

```
PROGRAM p104
!-------------------------------------------------------------------------
! Program 10.4: Eigenvalue analysis of an elastic solid in plane strain
!               using 4-node rectangular quadrilaterals. Lanczos Method.
!               Lumped mass. Element by element formulation.
!               Mesh numbered in x- or y-direction.
!-------------------------------------------------------------------------
 USE main; USE geom; iMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,iflag=-1,iters,jflag,k,lalfa,leig,lp=6,lx,lz,ndim=2,ndof, &
   neig=0,nels,neq,nip=4,nmodes,nn,nod=4,nodof=2,nprops=3,np_types,nr,    &
   nst=3,nxe,nye
 REAL(iwp)::acc,det,el,er,one=1.0_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-------------------------- dynamic arrays-------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),jeig(:,:),nf(:,:),&
   nu(:),num(:)
 REAL(iwp),ALLOCATABLE::alfa(:),bee(:,:),beta(:),coord(:,:),dee(:,:),     &
   del(:),der(:,:),deriv(:,:),diag(:),ecm(:,:),eig(:),fun(:),g_coord(:,:),&
   jac(:,:),km(:,:),mm(:,:),points(:,:),prop(:,:),ua(:),udiag(:),va(:),   &
   vdiag(:),v_store(:,:),weights(:),w1(:),x(:),x_coords(:),y(:,:),        &
   y_coords(:),z(:,:)
!---------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,nod,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye); ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),g_coord(ndim,nn),    &
   coord(nod,ndim),fun(nod),jac(ndim,ndim),weights(nip),g_num(nod,nels),  &
   der(ndim,nod),deriv(ndim,nod),bee(nst,ndof),num(nod),km(ndof,ndof),    &
   g(ndof),g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),eig(leig),x(lx),   &
   del(lx),nu(lx),jeig(2,leig),alfa(lalfa),beta(lalfa),z(lz,leig),        &
   prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 READ(10,*)nmodes,el,er,lalfa,leig,lx,lz,acc
!------------------loop the elements to set up global arrays--------------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END do elements_1
 WRITE(11,'(A,I5,A)')" There are",neq," equations"
 ALLOCATE(ua(0:neq),va(0:neq),vdiag(0:neq),diag(0:neq),udiag(0:neq),      &
   v_store(0:neq,lalfa),w1(0:neq),y(0:neq,leig))
 ua=zero; va=zero; eig=zero; jeig=0; x=zero; del=zero; nu=0; alfa=zero
 beta=zero; diag=zero; udiag=zero; w1=zero; y=zero; z=zero
 CALL sample(element,points,weights)
!--------------- element stiffness integration and assembly---------------
 elements_2: DO iel=1,nels
```

```
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   g=g_g(:,iel); km=zero; mm=zero
   integrating_pts_1: DO i=1,nip
     CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); call beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
     CALL ecmat(ecm,fun,ndof,nodof)
     mm=mm+ecm*det*weights(i)*prop(3,etype(iel))
   END DO integrating_pts_1
   DO i=1,ndof; diag(g(i))=diag(g(i))+SUM(mm(i,:)); END DO
 END DO elements_2
!-----------------------------find eigenvalues---------------------------
 diag=one/SQRT(diag); diag(0)=zero ! diag holds l**(-1/2)
 DO iters=1,lalfa
   CALL lancz1(neq,el,er,acc,leig,lx,lalfa,lp,iflag,ua,va,eig,jeig,neig,x, &
     del,nu,alfa,beta,v_store)
   IF(iflag==0)EXIT
   IF(iflag>1)THEN
    WRITE(11,'(A,I5)')" Lancz1 is signalling failure, with iflag = ",iflag
    STOP
   END IF
!----- iflag = 1 therefore form u + a * v  ( done element by element )----
   vdiag=va; vdiag=vdiag*diag ! vdiag is l**(-1/2).va
   udiag=zero; vdiag(0)=zero
   elements_3: DO iel=1,nels
     g=g_g(:,iel); udiag(g)=udiag(g)+MATMUL(km,vdiag(g))
   END DO elements_3
   udiag=udiag*diag; ua=ua+udiag
 END DO
!-------------- iflag = 0 therefore write out the spectrum ---------------
 WRITE(11,'(A,I4,A/)')" It took ",iters," iterations"
 WRITE(11,'(3(A,E12.4))')" Eigenvalues in the range",el," to",er," are:"
 WRITE(11,'(6E12.4)')eig(1:neig)
!  calculate the eigenvectors
 IF(neig>10)neig=10
 CALL lancz2(neq,lalfa,lp,eig,jeig,neig,alfa,beta,lz,jflag,y,w1,z,v_store)
!------------------if jflag is zero  calculate the eigenvectors ----------
 IF(jflag==0)THEN
   DO i=1,nmodes
     udiag(:)=y(:,i)
     udiag=udiag*diag
     WRITE(11,'(" Eigenvector number",I4," is:")')i
     WRITE(11,'(6E12.4)')udiag(1:)/MAXVAL(udiag(1:))
   END DO
 ELSE
! lancz2 fails
   WRITE(11,'(A,I5)')" Lancz2 is signalling failure with jflag = ",jflag
 END IF
STOP
END PROGRAM p104
```

New dynamic real arrays:

`vdiag`	used in element-by-element products

```
 There are   20 equations
 It took   22 iterations

 Eigenvalues in the range  0.0000E+00 to  0.5000E+01 are:
  0.4595E-02  0.1053E+00  0.1529E+00  0.5169E+00  0.1226E+01  0.1288E+01
  0.2058E+01  0.2352E+01  0.2415E+01  0.2710E+01  0.2719E+01  0.3114E+01
  0.3125E+01  0.3188E+01  0.3381E+01  0.3665E+01  0.3960E+01  0.4211E+01
  0.4962E+01
 Eigenvector number   1 is:
 -0.7195E-01  0.6932E-01  0.7195E-01  0.6932E-01 -0.1224E+00  0.2362E+00
  0.1224E+00  0.2362E+00 -0.1526E+00  0.4662E+00  0.1526E+00  0.4662E+00
 -0.1662E+00  0.7285E+00  0.1662E+00  0.7285E+00 -0.1695E+00  0.1000E+01
  0.1695E+00  0.1000E+01
 Eigenvector number   2 is:
 -0.2850E+00  0.4859E+00  0.2850E+00  0.4859E+00 -0.1171E+00  0.1000E+01
  0.1171E+00  0.1000E+01  0.2961E+00  0.9145E+00 -0.2961E+00  0.9145E+00
  0.6568E+00  0.8659E-01 -0.6568E+00  0.8659E-01  0.7873E+00 -0.1146E+01
 -0.7873E+00 -0.1146E+01
 Eigenvector number   3 is:
  0.3090E+00 -0.7493E-07  0.3090E+00 -0.1234E-06  0.5878E+00  0.8338E-07
  0.5878E+00  0.5351E-07  0.8090E+00  0.8965E-08  0.8090E+00  0.8734E-07
  0.9511E+00 -0.3425E-07  0.9511E+00 -0.3034E-09  0.1000E+01  0.7486E-08
  0.1000E+01 -0.1188E-06
 Eigenvector number   4 is:
 -0.2413E+00  0.9151E+00  0.2413E+00  0.9151E+00  0.4612E+00  0.8557E+00
 -0.4612E+00  0.8557E+00  0.5141E+00 -0.3742E+00 -0.5141E+00 -0.3742E+00
 -0.4120E+00 -0.6579E+00  0.4120E+00 -0.6579E+00 -0.1000E+01  0.7050E+00
  0.1000E+01  0.7050E+00
 Eigenvector number   5 is:
  0.2575E+00  0.9693E+00 -0.2575E+00  0.9693E+00  0.8105E+00 -0.2304E+00
 -0.8105E+00 -0.2304E+00 -0.3806E+00 -0.5583E+00  0.3806E+00 -0.5583E+00
 -0.6787E-01  0.6923E+00  0.6787E-01  0.6923E+00  0.1000E+01 -0.2476E+00
 -0.1000E+01 -0.2476E+00
```

Figure 10.12 Results from Program 10.4 example

The Lanczos algorithm relies on iterations which involve matrix–vector multiplication followed by vector addition. In the previous program, the matrix–vector product was performed using library subroutine `banmul` which took account of the storage strategy used to hold the global stiffness matrix `kb`.

Program 10.4 performs the same operations, but without the need to assemble and store a global stiffness matrix. The matrix–vector product described above can be achieved using element-by-element products as has been demonstrated in earlier programs in this book (e.g. Program 5.5).

The example and data are exactly the same as in Program 10.3 (Figure 10.10). The results are listed as Figure 10.12 which essentially reproduce the results from Program 10.2 which also assumed lumped mass. The Lanczos process took 23 iterations to converge in this case. Note the economy in storage requirements compared with Program 10.2.

Glossary of variable names used in Chapter 10

Scalar integers:

`i`	simple counter
`idiag`	skyline bandwidth
`iel`	simple counter

`ifail`	warning flag from `bisect` subroutine
`iflag`	no start vector specified, set to -1
`iters`	number of Lanczos iterations
`iwp`	`SELECTED_REAL_KIND(15)`
`j`	simple counters
`jflag`	equals zero if eigenvectors computed properly
`k`	simple counters
`lalfa`	Lanczos iteration ceiling
`leig`	maximum number of eigenvalues in range
`lp`	unit number for diagnostic messages
`lx`	set to at least `3*leig`
`lz`	holds first dimension of array `z`
`nband`	bandwidth of upper triangle
`ndim`	bandwidth of upper triangle
`ndof`	number of degrees of freedom per element
`neig`	holds the number of eigenvalues in `eig`
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integrating points
`nmodes`	number of eigenvectors required
`nn`	number of nodes in the mesh
`nod`	number of nodes per elements
`nodof`	number of degrees of freedom per node
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes
`nst`	number of stress (strain) terms
`nxe`	number of elements in the x-direction
`nye`	number of elements in the y-direction

Scalar reals:

`acc`	convergence tolerance relative to largest eigenvalue
`area`	element area
`det`	determinant of the Jacobian matrix
`d12`	set to 12.0
`el`	lower limit of eigenvalue spectrum
`er`	upper limit of eigenvalue spectrum
`etol`	eigenvalue tolerance set to 1×10^{-30}
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`pt5`	set to 0.5
`zero`	set to 0.0

Scalar character:

`element`	element type

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`jeig`	used to get the eigenvectors
`nf`	nodal freedom matrix
`nu`	working array
`num`	element node number vector

Dynamic real arrays:

`alfa`	working array holding Lanczos tridiagonal matrix
`bee`	strain-displacement matrix
`beta`	working array holding Lanczos tridiagonal matrix
`coord`	element nodal coordinates
`dee`	stress–strain matrix
`del`	working array
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`diag`	global lumped mass vector
`ecm`	Gauss point contribution to consistent mass matrix
`eig`	holds the computed eigenvalues in increasing order
`ell`	element lengths vector
`fun`	shape functions
`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`kb`	global stiffness matrix
`kdiag`	diagonal term location vector
`kh`	element stiffness matrix
`km`	element stiffness matrix
`ku`	global stiffness matrix
`kv`	global stiffness matrix
`mb`	global mass matrix
`mm`	element mass matrix
`points`	integrating point local coordinates
`prop`	element properties matrix
`rrmass`	reciprocal square rooted global lumped mass matrix
`ua`	working space vector
`udiag`	working space vector or eigenvector (in Lanczos)
`va`	working space vector
`vdiag`	used in element-by-element products
`v_store`	holds the Lanczos vectors
`weights`	weighting coefficients
`w1`	working space vector
`x`	working space vector
`x_coords`	x-coordinates of mesh layout

`y`	working space vector
`y_coords`	y-coordinates of mesh layout
`z`	working space vector

10.2 Exercises

1. Use Program 10.1 to evaluate the lowest two natural frequencies of the beam shown in Figure 10.13 with the boundary conditions.

   a) Pinned at both ends.

   b) Fixed at both ends.

   c) Fixed at one end and pinned at the other.

   d) Fixed at one end and free at the other.

   (Ans: Using 8 beam elements of equal length, "exact" solutions in parentheses.

   a) $\omega_1 = 223(225)\ \text{s}^{-1}$, $\omega_2 = \ 877(899)\ \text{s}^{-1}$

   b) $\omega_1 = 506(510)\ \text{s}^{-1}$, $\omega_2 = 1364(1405)\ \text{s}^{-1}$

   c) $\omega_1 = 349(351)\ \text{s}^{-1}$, $\omega_2 = 1107(1138)\ \text{s}^{-1}$

   d) $\omega_1 = \ 79(80)\ \text{s}^{-1}$, $\omega_2 = \ 480(502)\ \text{s}^{-1}$)

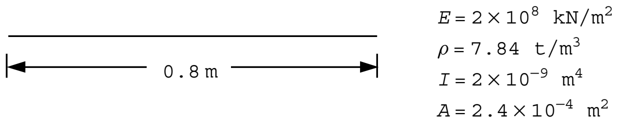

0.8 m
$E = 2 \times 10^8$ kN/m^2
$\rho = 7.84$ t/m^3
$I = 2 \times 10^{-9}$ m^4
$A = 2.4 \times 10^{-4}$ m^2

Figure 10.13

2. Repeat question 1 using Program 10.2 with 8-node elements.

   (Ans: Using a row of 8 square (0.1×0.1) 8-node elements with $E = 4800.0$, $\nu = 0.0$ and $\rho = 0.01882$, "exact" solutions in parentheses.

   a) $\omega_1 = 219(225)\ \text{s}^{-1}$, $\omega_2 = \ 824(899)\ \text{s}^{-1}$

   b) $\omega_1 = 474(510)\ \text{s}^{-1}$, $\omega_2 = 1196(1405)\ \text{s}^{-1}$

   c) $\omega_1 = 335(351)\ \text{s}^{-1}$, $\omega_2 = 1007(1138)\ \text{s}^{-1}$

   d) $\omega_1 = \ 80(80)\ \text{s}^{-1}$, $\omega_2 = \ 462(502)\ \text{s}^{-1}$)

3. Use Program 10.2 with 4-node elements to estimate the first two axial natural frequencies and mode shapes of the rod shown in Figure 10.14.

   (Ans: Using a row of 5 square (0.06×0.06) 4-node elements with $E = 2.1 \times 10^6$, $\nu = 0.0$ and $\rho = 0.0783$, "exact" solutions in parentheses. Axial deformation appear in the third and sixth modes.

$\omega_3 = 2.70 \times 10^4 (2.70 \times 10^4)$ s^{-1}, $[0.0 \ \ 0.31 \ \ 0.59 \ \ 0.81 \ \ 0.95 \ \ 1.00]^T$
$\omega_6 = 7.84 \times 10^4 (8.10 \times 10^4)$ s^{-1}, $[0.0 \ \ 0.81 \ \ 0.95 \ \ 0.31 \ \ -0.59 \ \ -1.00]^T$)

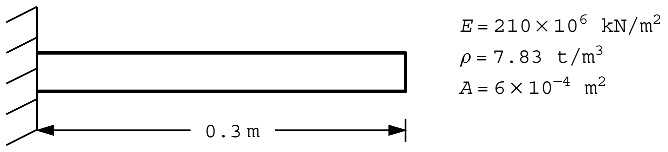


Figure 10.14

4. Repeat question 3 using rod elements. Program 10.1 is easily modified to analyse axial vibrations of rods. Make the following changes:

   a) In the declarations change `ndof=4` to `ndof=2` and `nodof=2` to `nodof=1`.
   b) In the `elements_2:` loop delete the statements that assign values to `mm(2,2)=...` and `mm(4,4)=...` and then change `mm(3,3)` to `mm(2,2)`.
   c) Change routine `beam_km` to `rod_km`.

   The data read into `prop` will now be `ea` (instead of `ei`) and `rhoa`.
   (Ans: Using 5 equal rod elements with $EA = 1.26 \times 10^5$ and $\rho A = 4.70 \times 10^{-3}$ answers exactly the same as for question 3.)

5. Using the modified program from the previous question determine the first two natural frequencies and eigenvectors for the stepped bar shown in Figure 10.15.
   (Ans: $\omega_1 = 2.46 \times 10^4$ s^{-1}, $[0.0 \ \ 0.50 \ \ 0.82 \ \ 0.95 \ \ 1.00]^T$
   $\omega_2 = 5.94 \times 10^4$ s^{-1}, $[0.0 \ \ 0.68 \ \ -0.08 \ \ -0.74 \ \ -1.00]^T$)

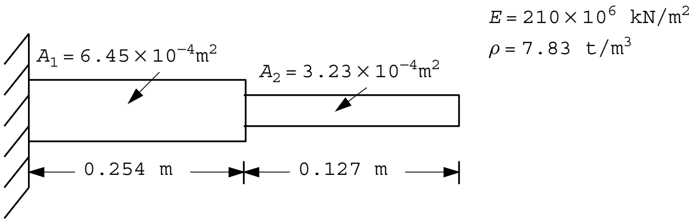

$E = 210 \times 10^6$ kN/m^2
$\rho = 7.83$ t/m^3
$A_1 = 6.45 \times 10^{-4}$m^2
$A_2 = 3.23 \times 10^{-4}$m^2
0.254 m
0.127 m

Figure 10.15

6. A vital attribute of an element "stiffness matrix" is that it should possess the right number of "rigid body modes", that is zero eigenvalues of the stiffness matrix. Test the 2-node elastic rod element stiffness matrix and prove that it has only one zero eigenvalue, corresponding to one rigid body mode (translation). If the rod has length L, cross-sectional area A, modulus E and mass per unit length ρ, calculate its non-zero eigenvalue and hence natural frequency of free vibration assuming both lumped and consistent mass. Compare with the "exact" value of $\pi/L\sqrt{E/\rho}$.
   (Ans: Lumped:$2/L\sqrt{E/\rho}$, Consistent: $2\sqrt{3}/L\sqrt{E/\rho}$)

7. Show how dynamic equilibrium of a multi-degree of freedom system vibrating at a resonant frequency leads to an equation of the form:

$$[\mathbf{K}_m]\{\mathbf{X}\} = \omega^2[\mathbf{M}_m]\{\mathbf{X}\}$$

where $[\mathbf{K}_m]$, $[\mathbf{M}_m]$ = system stiffness and mass matrices, $\{\mathbf{X}\}$ = displacement amplitudes, ω^2 = angular frequency. Describe a method for reducing this equation to a standard eigenvalue equation of the form:

$$[\mathbf{A}]\{\mathbf{Z}\} = \omega^2\{\mathbf{Z}\}$$

where $[\mathbf{A}]$ is symmetrical.

References

Chopra AK 1995 *Dynamics of Structures*. Prentice–Hall, Englewood Cliffs, N.J.

Cook RD, Malkus DS and Plesha ME 1989 *Concepts and Applications of Finite Element Analysis*, 3rd edn. John Wiley & Sons, Chichester, New York.

HSL 2002 *A Collection of Fortran Codes for Large Scientific Computation*. See `http://www.cse.clrc.ac.uk/nag/hsl/`.

Parlett BN and Reid JK 1981 Tracking the progress of the Lanczos algorithm for large symmetric eigenproblems. *IMA J Numer Anal* **1**, 135–155.

Smith IM 1977 Transient phenomena of offshore foundations. *Numerical Methods in Offshore Engineering*. John Wiley & Sons, Chichester, New York, Chapter 14, pp. 483–513.

Smith IM 1984 Adaptability of truly modular software. *Eng Comput* **1**(1), 25–35.

Smith IM and Heshmati EE 1983 Use of a Lanczos algorithm in dynamic analysis of structures. *Earthquake Eng Struct Dyn* **11**(4), 585–588.

11

Forced Vibrations

11.1 Introduction

In the previous chapter, programs were described which enable the calculation of the intrinsic dynamic properties of systems, namely their undamped natural frequencies and mode shapes. The next stage in a dynamic analysis is usually the calculation of the response of the system to an imposed time dependent disturbance. This chapter describes seven programs which enable such calculations to be made.

The types of equations to be solved were derived early in the book (e.g. 2.13). After semi-discretisation in space using finite elements, the resulting matrix equations are typified by (2.17), a set of second order ordinary differential equations in time. On inclusion of damping, the relevant equations become (3.118) and Section 3.13 describes the principles behind the various solution procedures used below.

Program 11.1 describes forced vibration analysis of elastic slender beam structures using direct integration in the "time domain". Programs 11.2, 11.3, and 11.4 describe forced vibration analyses of planar 2D elastic solids. Program 11.2 works in the "frequency domain" using the Modal Superposition Method, and Programs 11.3 and 11.4 work in the "time domain" utilising two different implicit time-marching algorithms. Program 11.5 illustrates a "mixed" time-marching scheme for 2D analysis in which some parts of the mesh are integrated "explicitly" and others "implicitly". Program 11.6 repeats the algorithm of Program 11.3 using a "mesh free" approach with a preconditioned conjugate gradient solver. Program 11.7 uses a fully explicit time marching scheme to analyse a 2D elastic–plastic material.

Programs 11.3 and 11.7 have parallel counterparts described in Chapter 12.

Programming the Finite Element Method 4e I. M. Smith and D. V. Griffiths
© 2004 John Wiley & Sons, Ltd ISBNs: 0-470-84970-3 (PBK); 0-470-84969-X (PPC)

Program 11.1 Forced vibration analysis of elastic beams using 2-node beam elements. Consistent mass. Newmark time stepping.

```
PROGRAM p111
!-------------------------------------------------------------------------
! Program 11.1 Forced vibration analysis of elastic beams using 2-node
!              beam elements. Consistent mass. Newmark time stepping.
!-------------------------------------------------------------------------
 USE main; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,iel,j,k,lnode,lsense,ndof=4,nels,neq,nlf,nn,nod=2,nodof=2,nof,&
   nlfp,nprops=2,np_types,nr,nstep
 REAL(iwp)::beta,dtim,fk,fm,f1,f2,gamma,one=1.0_iwp,pt5=0.5_iwp,          &
   zero=0.0_iwp
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),kdiag(:),lf(:),lp(:),nf(:,:),&
   node(:),num(:),sense(:)
 REAL(iwp),ALLOCATABLE::a(:),acc(:,:),al(:,:),a1(:),b1(:),cv(:),d(:),     &
   dis(:,:),ell(:),kd(:),km(:,:),kp(:),kv(:),mc(:),mm(:,:),mv(:),         &
   prop(:,:),rl(:),rt(:),v(:),vc(:),vel(:,:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nels,np_types; nn=nels+1
 ALLOCATE(nf(nodof,nn),km(ndof,ndof),mm(ndof,ndof),num(nod),g(ndof),      &
   prop(nprops,np_types),ell(nels),g_g(ndof,nels),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)ell,dtim,beta,gamma,fm,fk
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag(neq),a1(0:neq),b1(0:neq),vc(0:neq),kd(0:neq),a(0:neq),    &
   d(0:neq),v(0:neq)); kdiag=0
!-----------------------loop the elements to find global array sizes------
 elements_1: DO iel=1,nels
   num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
   CALL fkdiag(kdiag,g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),cv(kdiag(neq)),mv(kdiag(neq)),mc(kdiag(neq)),    &
   kp(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 kv=zero; mv=zero
!-----------------------global stiffness and mass matrix assembly---------
 elements_2: DO iel=1,nels
   CALL beam_km(km,prop(1,etype(iel)),ell(iel))
   CALL beam_mm(mm,prop(2,etype(iel)),ell(iel)); g=g_g(:,iel)
   CALL fsparv(kv,km,g,kdiag); CALL fsparv(mv,mm,g,kdiag)
 END DO elements_2; mc=mv
!-----------------------initial conditions, and load functions------------
 d=zero; v=zero   !  alternatively READ(10,*)d(1:),v(1:)
 READ(10,*)nlf; ALLOCATE(lf(nlf))
 DO k=1,nlf
   READ(10,*)lnode,lsense,nlfp; ALLOCATE(rt(nlfp),rl(nlfp))
   lf(k)=nf(lsense,lnode); READ(10,*)(rt(j),rl(j),j=1,nlfp)
   IF(k==1)THEN
     nstep=NINT((rt(nlfp)-rt(1))/dtim)+1; ALLOCATE(al(nstep,nlf))
   END IF; CALL interp(k,dtim,rt,rl,al,nstep); DEALLOCATE(rt,rl)
 END DO
 f1=beta*dtim**2; f2=beta*dtim; cv=fm*mv+fk*kv; kp=mv/f1+gamma*cv/f2+kv
 CALL sparin(mc,kdiag); CALL sparin(kp,kdiag); a=zero; a(lf(:))=al(1,:)
 CALL linmul_sky(cv,v,vc,kdiag); CALL linmul_sky(kv,d,kd,kdiag); a=a-vc-kd
 CALL spabac(mc,a,kdiag); READ(10,*)nof
 ALLOCATE(node(nof),sense(nof),lp(nof),dis(nstep,nof),vel(nstep,nof),     &
   acc(nstep,nof)); READ(10,*)(node(i),sense(i),i=1,nof)
 DO i=1,nof; lp(i)=nf(sense(i),node(i)); END DO
 dis(1,:)=d(lp); vel(1,:)=v(lp); acc(1,:)=a(lp)
!-----------------------time stepping loop----------------------------------
 DO j=2,nstep
   a1=d/f1+v/f2+a*(pt5/beta-one)
   b1=gamma*d/f2-v*(one-gamma/beta)-dtim*a*(one-pt5*gamma/beta)
   CALL linmul_sky(mv,a1,vc,kdiag); CALL linmul_sky(cv,b1,kd,kdiag)
   d=vc+kd; d(lf(:))=d(lf(:))+al(j,:); CALL spabac(kp,d,kdiag)
   v=gamma*d/f2-b1; a=d/f1-a1
   dis(j,:)=d(lp); vel(j,:)=v(lp); acc(j,:)=a(lp)
 END DO
 DO i=1,nof
   WRITE(11,'(/,2(A,I3))')" Output at node",node(i),", sense",sense(i)
   WRITE(11,'(A)')"    time        disp        velo        accel"
   DO j=1,nstep; WRITE(11,'(4E12.4)')(j-1)*dtim,dis(j,i),vel(j,i),acc(j,i)
   END DO
 END DO
STOP
END PROGRAM p111
```

Scalar integers:

`i`	simple counter
`iel`	simple counter
`iwp`	`SELECTED_REAL_KIND(15)`
`j`	simple counter
`k`	simple counter
`lnode`	loaded node number
`lsense`	sense of freedom to be loaded at node `lnode`
`ndof`	number of degrees of freedom per element
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nlfp`	number of load function points
`nln`	number of loaded freedoms
`nn`	number of nodes in the mesh
`nod`	number of nodes per elements
`nodof`	number of degrees of freedom per node
`nof`	number of output freedoms
`nprops`	number of material properties
`np_types`	number of different property types
`nr`	number of restrained nodes
`nstep`	number of calculation time steps

Scalar reals:

`beta`	Newmark time stepping parameter
`dtim`	calculation time step
`fk`	Rayleigh damping parameter on stiffness
`fm`	Rayleigh damping parameter on mass
`f1`	temporary working variable
`f2`	temporary working variable

`gamma`	Newmark time stepping parameter
`one`	set to 1.0
`pt5`	set to 0.5
`zero`	set to 0.0

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`kdiag`	diagonal term locations
`lf`	vector holding loaded freedoms
`lp`	vector holding output freedoms
`nf`	nodal freedom matrix
`node`	output nodes
`num`	element node number vector
`sense`	sense of output nodes

Dynamic real arrays:

`a`	accelerations
`acc`	accelerations at output freedoms
`al`	array holding all loading values at each calculation step
`a1`	temporary working vector
`b1`	temporary working vector
`cv`	damping matrix
`d`	displacements
`dis`	displacements at output freedoms
`ell`	element lengths vector
`kd`	vector used to set up initial accelerations
`km`	beam element stiffness matrix
`kp`	modified global "stiffness" matrix
`kv`	global stiffness matrix
`mc`	global mass matrix used to set up initial accelerations
`mm`	element consistent mass matrix
`mv`	global consistent mass matrix
`prop`	element properties matrix
`rl`	input load function load values
`rt`	input load function time values
`v`	velocities
`vc`	vector used to set up initial accelerations
`vel`	velocities at output freedoms

This program computes the response of a string of beam elements to a combination of time-dependent applied nodal loads. The program allows for (Rayleigh) damping (Section 3.13) and uses a consistent mass matrix (2.30). Time stepping is achieved using a direct Newmark method involving time stepping parameters β and γ, the values of which determine the accuracy and stability characteristics of the algorithm (see e.g. Bathe, 1996).

For the special case of $\beta = 1/4$ and $\gamma = 1/2$, the method is identical to the Crank–Nicolson $\theta = 0.5$ method described in Section 3.13.2.

Starting from the assembled form of (3.118), and using a "dot" notation to signify time derivatives, we have

$$[\mathbf{K}_m]\{\mathbf{U}\} + [\mathbf{C}_m]\{\dot{\mathbf{U}}\} + [\mathbf{M}_m]\{\ddot{\mathbf{U}}\} = \{\mathbf{F}\} \tag{11.1}$$

with known initial conditions on displacements and velocities, $\{\mathbf{U}\}_0$ and $\{\dot{\mathbf{U}}\}_0$.

Let $\{\mathbf{F}\}_i$, $\{\mathbf{U}\}_i$, $\{\dot{\mathbf{U}}\}_i$ and $\{\ddot{\mathbf{U}}\}_i$ represent conditions at time $t = i\Delta t$, where $i = 0, 1, 2, \dots,$ `nstep`.

Assuming that $[\mathbf{M}_m]$, $[\mathbf{C}_m]$, $[\mathbf{K}_m]$, β, γ and Δt are constant, and that $\{\mathbf{F}\}_i$ is known for all i, the following algorithm is used to obtain the values of $\{\mathbf{U}\}_i$, $\{\dot{\mathbf{U}}\}_i$ and $\{\ddot{\mathbf{U}}\}_i$ for all $i > 0$.

1) Compute:
$$[\mathbf{K}'] = [\mathbf{K}_m] + \frac{\gamma}{\beta\Delta t}[\mathbf{C}_m] + \frac{1}{\beta(\Delta t)^2}[\mathbf{M}_m]$$

2) Factorise $[\mathbf{K}']$ to facilitate step 8.

3) Solve the linear equations: $[\mathbf{M}_m]\{\ddot{\mathbf{U}}\}_0 = \{\mathbf{F}\}_0 - [\mathbf{C}_m]\{\dot{\mathbf{U}}\}_0 - [\mathbf{K}_m]\{\mathbf{U}\}_0$

4) Set $i = 0$

5) Compute:
$$\{\mathbf{A}\}_i = \frac{1}{\beta(\Delta t)^2}\{\mathbf{U}\}_i + \frac{1}{\beta\Delta t}\{\dot{\mathbf{U}}\}_i + \left(\frac{1}{2\beta} - 1\right)\{\ddot{\mathbf{U}}\}_i$$

6) Compute:
$$\{\mathbf{B}\}_i = \frac{\gamma}{\beta\Delta t}\{\mathbf{U}\}_i - \left(1 - \frac{\gamma}{\beta}\right)\{\dot{\mathbf{U}}\}_i - \left(1 - \frac{\gamma}{2\beta}\right)\Delta t\{\ddot{\mathbf{U}}\}_i$$

7) Compute: $\{\mathbf{F}'\}_{i+1} = \{\mathbf{F}\}_{i+1} + [\mathbf{M}_m]\{\mathbf{A}\}_i + [\mathbf{C}_m]\{\mathbf{B}\}_i$

8) Solve the linear equations: $[\mathbf{K}']\{\mathbf{U}\}_{i+1} = \{\mathbf{F}'\}_{i+1}$

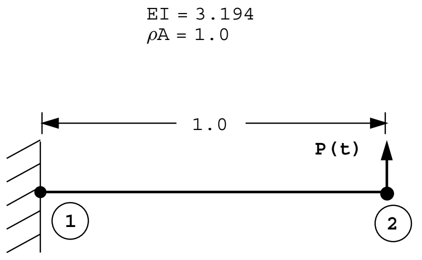
9) Compute:
$$\{\dot{\mathbf{U}}\}_{i+1} = \frac{\gamma}{\beta\Delta t}\{\mathbf{U}\}_{i+1} - \{\mathbf{B}\}_i$$

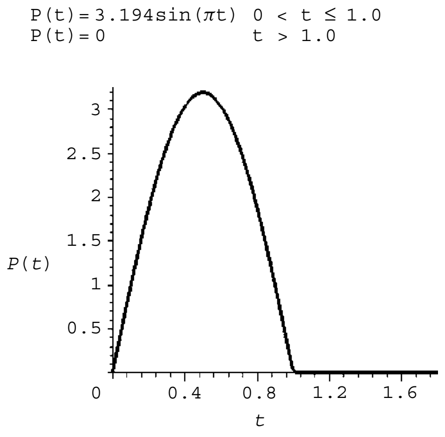
10) Compute:
$$\{\ddot{\mathbf{U}}\}_{i+1} = \frac{1}{\beta(\Delta t)^2}\{\mathbf{U}\}_{i+1} - \{\mathbf{A}\}_i$$

11) Increment i and repeat from step 5

Subroutine `beam_mm` (see e.g. Program 4.6) forms the beam element consistent mass matrix and subroutine `interp` takes the input load/time function data points and interpolates linearly to give load/time function values at the resolution of the calculation time step.

The example and data shown in Figure 11.1 are of a cantilever of unit length modelled with a single beam element, subjected to a tip loading given by a half-sine pulse with an





```
nels  np_types
1       1

prop(ei,rhoa)
3.194  1.0

etype (not needed)

ell
1.0

dtim  beta  gamma  fm   fk
0.05  0.25   0.5   0.0  0.0

nr,(k,nf(:,k),i=1,nr)
1
1 0 0

nlf
1

lnode  lsense
2        1

nlfp, (rt(j),rl(j),j=1,nlfp)
12
0.000  0.000   0.100  0.987   0.200  1.877   0.300  2.584
0.400  3.038   0.500  3.194   0.600  3.038   0.700  2.584
0.800  1.877   0.900  0.987   1.000  0.000   1.800  0.000

nof,(node(i),sense(i),i=1,nof)
1
2  1
```

Figure 11.1 Mesh and data for Program 11.1 example

amplitude of *EI* and a duration equal to T, the fundamental period of the cantilever. The properties of the beam are given by an *EI* of 3.194 and a mass per unit length ρA of 1.0. The fundamental frequency of the beam from equation (10.4) is therefore given by $\omega = 3.516\sqrt{3.194} = 6.284$, and the fundamental period by $T = 2\pi/\omega = 1.00$.

The first line of data reads the number of elements `nels` followed by the number of property groups and the properties *EI* and ρA. The next line reads the element lengths `ell` and the next line gives the time-stepping data. In this case the calculation time step is `dtim=0.05` and the conventional Newmark time-stepping parameters are used, namely

$\beta = 0.25$ and $\gamma = 0.5$ (read as `beta` and `gamma` respectively). The beam is undamped, thus the Rayleigh damping parameters (`fm` and `fk`) are both set to zero. The boundary condition data indicate one restrained node at the built-in end of the cantilever, where both freedoms are restrained. There is one loaded freedom (`nlf=1`) in this example at node 2, sense 1 (the translational freedom). The sinusoidal loading function to be applied has been input using a total of 12 coordinates comprising 11 coordinates at 0.1 s intervals up to 1 s, followed by a "quiet zone" involving a single interval of 0.8 s up to 1.8 s. The program linearly interpolates this load/time function at the calculation step length of 0.05 s for the Newmark algorithm. The final data gives the freedoms at which output is required. In this example, the displacement, velocity and acceleration under the load are of interest, so there is just one output required (`nof=1`), again at node 2 sense 1.

The output from Program 11.1 is shown in Figure 11.2 and a plot of the computed displacement/time history of the cantilever tip is shown in Figure 11.3. For comparison,

```
There are    2 equations and the skyline storage is    3

Output at node  2, sense  1
   time        disp        velo        accel
 0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
 0.5000E-01  0.1942E-02  0.7768E-01  0.3107E+01
 0.1000E+00  0.9511E-02  0.2251E+00  0.2788E+01
 0.1500E+00  0.2531E-01  0.4069E+00  0.4485E+01
 0.2000E+00  0.5285E-01  0.6946E+00  0.7026E+01
 0.2500E+00  0.9488E-01  0.9866E+00  0.4652E+01
 0.3000E+00  0.1497E+00  0.1207E+01  0.4159E+01
 0.3500E+00  0.2153E+00  0.1415E+01  0.4184E+01
 0.4000E+00  0.2892E+00  0.1541E+01  0.8437E+00
 0.4500E+00  0.3650E+00  0.1493E+01 -0.2780E+01
 0.5000E+00  0.4362E+00  0.1352E+01 -0.2843E+01
 0.5500E+00  0.4976E+00  0.1104E+01 -0.7080E+01
 0.6000E+00  0.5421E+00  0.6758E+00 -0.1005E+02
 0.6500E+00  0.5633E+00  0.1746E+00 -0.9996E+01
 0.7000E+00  0.5589E+00 -0.3524E+00 -0.1108E+02
 0.7500E+00  0.5261E+00 -0.9574E+00 -0.1312E+02
 0.8000E+00  0.4638E+00 -0.1534E+01 -0.9945E+01
 0.8500E+00  0.3757E+00 -0.1993E+01 -0.8412E+01
 0.9000E+00  0.2659E+00 -0.2397E+01 -0.7734E+01
 0.9500E+00  0.1393E+00 -0.2670E+01 -0.3187E+01
 0.1000E+01  0.4299E-02 -0.2729E+01  0.8097E+00
 0.1050E+01 -0.1284E+00 -0.2581E+01  0.5126E+01
 0.1100E+01 -0.2487E+00 -0.2229E+01  0.8928E+01
 0.1150E+01 -0.3455E+00 -0.1642E+01  0.1458E+02
 0.1200E+01 -0.4081E+00 -0.8625E+00  0.1658E+02
 0.1250E+01 -0.4308E+00 -0.4551E-01  0.1610E+02
 0.1300E+01 -0.4123E+00  0.7824E+00  0.1701E+02
 0.1350E+01 -0.3534E+00  0.1575E+01  0.1468E+02
 0.1400E+01 -0.2597E+00  0.2174E+01  0.9284E+01
 0.1450E+01 -0.1415E+00  0.2554E+01  0.5935E+01
 0.1500E+01 -0.9326E-02  0.2733E+01  0.1196E+01
 0.1550E+01  0.1243E+00  0.2614E+01 -0.5935E+01
 0.1600E+01  0.2452E+00  0.2221E+01 -0.9787E+01
 0.1650E+01  0.3422E+00  0.1660E+01 -0.1265E+02
 0.1700E+01  0.4067E+00  0.9181E+00 -0.1702E+02
 0.1750E+01  0.4310E+00  0.5512E-01 -0.1750E+02
 0.1800E+01  0.4132E+00 -0.7674E+00 -0.1540E+02
```

Figure 11.2 Results from Program 11.1 example

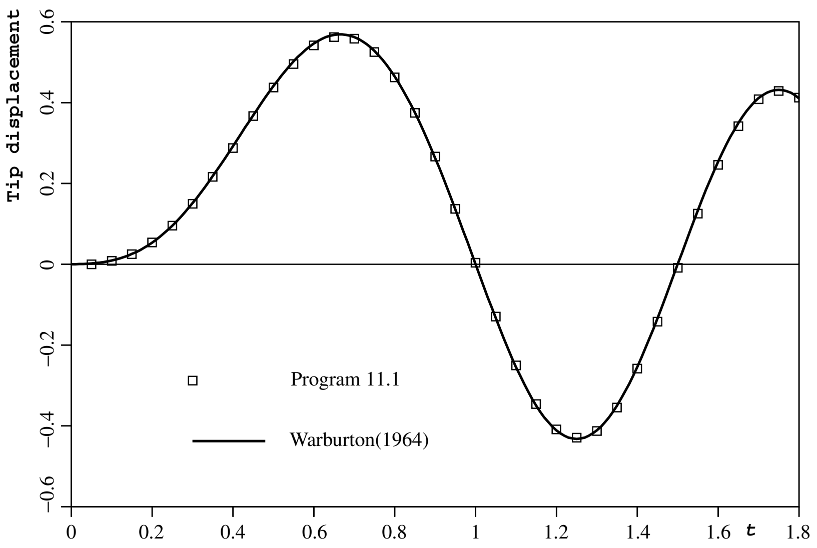


Figure 11.3 Computed tip displacement-time history from Program 11.1 example

the analytical solution (e.g. Warburton, 1964) for this problem during the loading phase when $0 \leq t \leq T$ is given by:

$$v(t) = 0.441 \sin \frac{\pi t}{T} - 0.216 \sin \frac{2\pi t}{T} \tag{11.2}$$

and for the free vibration phase when $t > T$ by,

$$v(t) = -0.432 \sin \{6.284(t - T)\} \tag{11.3}$$

It is clear from Figure 11.3 that this analytical solution is virtually indistinguishable from the computed result.

Program 11.2 Forced vibration analysis of an elastic solid in plane strain using 4- or 8-node rectangular quadrilaterals. Lumped mass. Mesh numbered in x- or y-direction. Modal superposition.

```
PROGRAM p112
!-------------------------------------------------------------------------
! Program 11.2 Forced vibration of an elastic solid in plane strain
!              using 4- or 8-node rectangular quadrilaterals. Lumped mass.
!              Mesh numbered in x- or y-direction. Modal superposition.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,idiag,iel,ifail,j,jj,k,nband,ndim=2,ndof,nels,neq,nmodes,nn,  &
   nod,nodof=2,npri,nprops=3,np_types,nr,nres,nstep,nxe,nye
 REAL(iwp)::aa,area,bb,dr,dtim,d4=4.0_iwp,etol=1.e-30_iwp,f,k1,k2,omega,  &
   one=1.0_iwp,penalty=1.0e20_iwp,time,two=2.0_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g (:,:),g_num(:,:),kdiag(:),nf(:,:),&
   num(:)
 REAL(iwp),ALLOCATABLE::bigk(:,:),coord(:,:),diag(:),g_coord(:,:),kh(:),  &
   km(:,:),ku(:,:),kv(:),mm(:,:),prop(:,:),rrmass(:),udiag(:),xmod(:),    &
   x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,nod,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ndof=nod*nodof
 ALLOCATE(nf(nodof,nn),g_coord(ndim,nn),coord(nod,ndim),g_num(nod,nels),  &
   num(nod),km(ndof,ndof),g(ndof),g_g(ndof,nels),x_coords(nxe+1),         &
   y_coords(nye+1),prop(nprops,np_types),etype(nels),mm(ndof,ndof))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,npri,nres,dr
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 READ(10,*)nmodes,omega
 ALLOCATE(diag(0:neq),udiag(0:neq),kdiag(neq),rrmass(0:neq),xmod(nmodes), &
   bigk(neq,nmodes))
!-------loop the elements to find nband and set up global arrays----------
   nband=0; kdiag=0
   elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
   IF(nband<bandwidth(g))nband=bandwidth(g)
 END DO elements_1
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 WRITE(11,'(A,I5,A,/,A,I5,/,A,I5)')" There are",neq," equations",         &
   " The half-bandwidth (including diagonal) is",nband+1,                 &
   " The skyline storage is",kdiag(neq)
   diag=zero; ku=zero; bigk=zero
!-----------------------element stiffness and mass assembly---------------
 ALLOCATE(ku(neq,nband+1),kv(kdiag(neq)),kh(kdiag(neq)))
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
   CALL rect_km(km,coord,prop(1,etype(iel)),prop(2,etype(iel)))
   CALL formku(ku,km,g)
   area=(MAXVAL(coord(:,1))-MINVAL(coord(:,1)))*                          &
        (MAXVAL(coord(:,2))-MINVAL(coord(:,2)))
   CALL elmat(area,prop(3,etype(iel)),mm); CALL formlump(diag,mm,g)
  END DO elements_2; rrmass(1:)=one/SQRT(diag(1:))
!-----------------------reduce to standard eigenvalue problem-------------
  DO i=1,neq; IF(i<=neq-nband)THEN; k=nband+1; ELSE; k=neq-i+1; END IF
    DO j=1,k; ku(i,j)=ku(i,j)*rrmass(i)*rrmass(i+j-1); END DO
  END DO
!-----------------------convert to skyline form---------------------------
 kh(1)=ku(1,1); k=1
 DO i=2,neq; idiag=kdiag(i)-kdiag(i-1)
   DO j=1,idiag; k=k+1; kh(k)=ku(i+j-idiag,1-j+idiag); END DO
 END DO
!-----------------------extract the eigenvalues---------------------------
```

```
  CALL bandred(ku,diag,udiag); ifail=1; CALL bisect(diag,udiag,etol,ifail)
!-----------------------extract the "mass normalised" eigenvectors--------
 DO i=1,nmodes
   kv=kh; kv(kdiag)=kv(kdiag)-diag(i); kv(1)=kv(1)+penalty
   udiag=zero; udiag(1)=kv(1)
   CALL sparin_gauss(kv,kdiag); CALL spabac_gauss(kv,udiag,kdiag)
   udiag=rrmass*udiag
   udiag(1:)=udiag(1:)/SQRT(SUM(udiag(1:)**2/rrmass(1:)**2))
   bigk(:,i)=udiag(1:)
 END DO; udiag=zero; time=zero
 WRITE(11,'(/A,I5)')" Result at node",nres
 WRITE(11,'(A)')"    time         load         x-disp      y-disp"
 WRITE(11,'(4E12.4)')time,COS(omega*time),udiag(nf(:,nres))
!------------------------time stepping loop-------------------------------
 DO jj=1,nstep; time=time+dtim
   DO i=1,nmodes; f=bigk(neq,i)
!-----------------------analytical solution for cosine loading------------
     k1=diag(i)-omega**2; k2=k1*k1+d4*omega**2*dr**2*diag(i)
     aa=f*k1/k2; bb=f*two*omega*dr*SQRT(diag(i))/k2
     xmod(i)=aa*COS(omega*time)+bb*SIN(omega*time)
   END DO
!-----------------------superpose the modes-------------------------------
   udiag(1:)=MATMUL(bigk,xmod(1:))
   IF(jj/npri*npri==jj)WRITE(11,'(4E12.4)')time,COS(omega*time),          &
     udiag(nf(:,nres))
 END DO
STOP
END program p112
```

New scalar integers:

`idiag`	skyline bandwidth
`ifail`	warning flag from `bisect` subroutine
`jj`	simple counter
`nband`	bandwidth of upper triangle
`ndim`	number of dimensions
`nmodes`	number of eigenvectors included in superposition
`npri`	output printed every `npri` time steps
`nres`	node number at which time history is to be printed
`nstep`	number of time steps required
`nxe`	number of elements in x-direction
`nye`	number of elements in y-direction

New scalar reals:

`aa`	working variable
`area`	element area
`bb`	working variable
`dr`	damping ratio
`d4`	set to 4.0
`etol`	eigenvalue tolerance set to 1×10^{-30}
`f`	force vector
`k1`	working variable

`k2`	working variable
`omega`	frequency of forcing term
`penalty`	set to 1×10^{20}
`time`	holds elapsed time t
`two`	set to 2.0

Scalar character:

`element`	element type

New Dynamic integer arrays:

`g_num`	global element node numbers matrix

New Dynamic real arrays:

`bigk`	eigenvector matrix
`coord`	element nodal coordinates
`diag`	global lumped mass vector
`g_coord`	nodal coordinates for all elements
`kh`	global stiffness vector
`km`	element stiffness matrix
`ku`	global stiffness matrix stored as upper triangle
`rrmass`	vector holding reciprocal of square root of lumped mass
`udiag`	transformed and untransformed eigenvectors
`xmod`	solutions to modal SDOF equations
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout

Since the basis of this method is the synthesis of the undamped natural modes of the vibrating system, it follows very naturally from the programs of the previous chapter. Indeed this program can be built up, with minor extensions, from Program 10.2. The method is described in Section 3.13.1.

The illustrative problem chosen for this program and the two that follow is shown in Figure 11.4 and is similar to the cantilever beam considered in Figure 10.6. The beam in

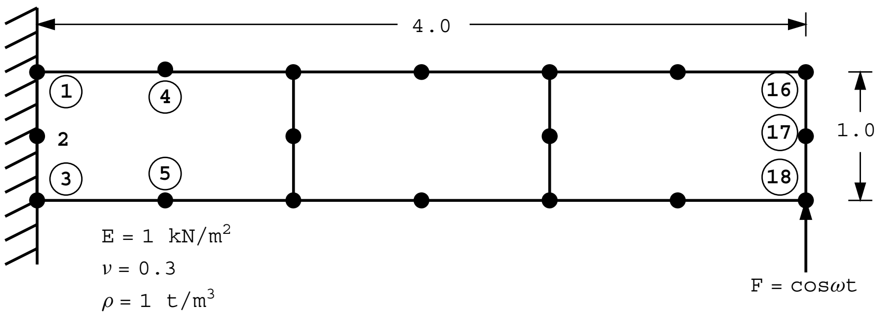
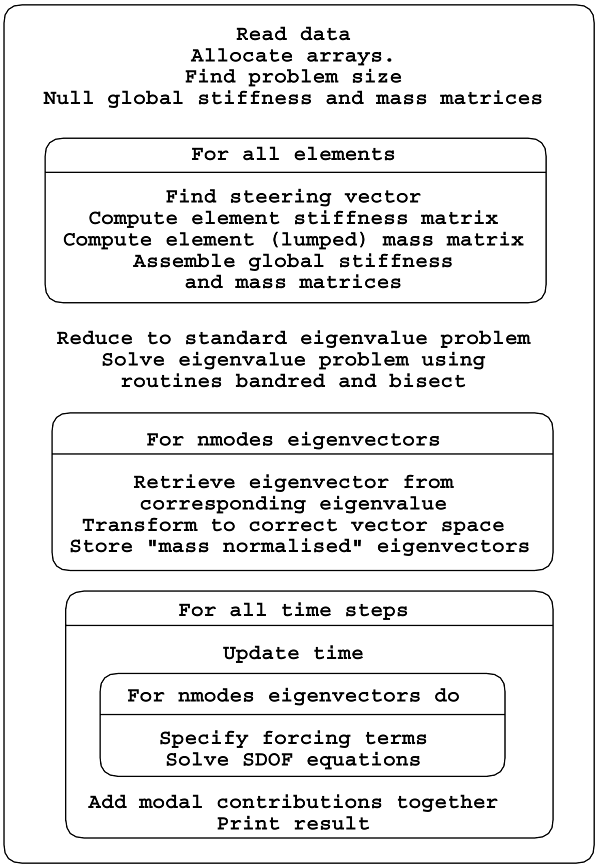


Figure 11.4 Mesh and data for Program 11.2 example (*Continued on page 476*)

```
nxe  nye  nod
3     1    8

np_types
1

prop(e,v,rho)
1.0  0.3  1.0

etype (not needed)

x_coords, y_coords
0.0 1.33333  2.66667  4.0
0.0  -1.0

dtim  nstep  npri  nres   dr
1.0    20     1     18   0.05

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

nmodes  omega
6        0.3
```

Figure 11.4 (*Continued from page 475*)



```
Read data
Allocate arrays.
Find problem size
Null global stiffness and mass matrices
For all elements
    Find steering vector
    Compute element stiffness matrix
    Compute element (lumped) mass matrix
    Assemble global stiffness
    and mass matrices
Reduce to standard eigenvalue problem
Solve eigenvalue problem using
routines bandred and bisect
For nmodes eigenvectors
    Retrieve eigenvector from
    corresponding eigenvalue
    Transform to correct vector space
    Store "mass normalised" eigenvectors
For all time steps
    Update time
    For nmodes eigenvectors do
        Specify forcing terms
        Solve SDOF equations
    Add modal contributions together
    Print result
```

Figure 11.5 Structure chart for Program 11.2

Figure 11.4 is subjected to a harmonic vertical force of $\cos \omega t$ at node 18. The damping ratio ζ (see equation (3.120)), called `dr` in the program is 0.05 or 5% applied to all modes of the system.

The forcing frequency ω (called `omega` in the program) is set at 0.3, which is deliberately chosen to be close to the second natural frequency of the undamped system (Note: $\omega_2 \approx 0.3$ from Figure 10.7), so at this frequency the influence of damping should be significant.

The structure of the program is essentially the same as Program 10.2 up to the end of the section headed "extract the mass normalised eigenvectors", however the current program uses a different eigenvector normalisation strategy that reduces the modal mass matrix to a unit matrix. It should be remembered that the eigenvectors first computed as `udiag` are those of the transformed problem and the true eigenvectors must be recovered prior to normalisation and storage in the eigenvector matrix `bigk`.

When the time stepping loop is entered, the cosine loading of a single degree of freedom system is introduced, and the modal contributions superposed at each degree of freedom. A structure chart for the modal superposition algorithm is given in Figure 11.5.

The example uses the first six eigenmodes, read as `nmodes=6` to synthesise the time response. Users are invited to examine the sensitivity of the response to different values of `nmodes` up to a maximum of 30 in this case. The output from the analysis is shown in Figure 11.6 and the response in the y-direction at node 18 is plotted in Figure 11.7.

```
There are    30  equations and the half-bandwidth is   15

Result at node   18
   time        load        x-disp      y-disp
 0.0000E+00  0.1000E+01  0.0000E+00  0.0000E+00
 0.1000E+01  0.9553E+00  0.2188E+02  0.2612E+02
 0.2000E+01  0.8253E+00  0.2825E+02  0.3727E+02
 0.3000E+01  0.6216E+00  0.3210E+02  0.4508E+02
 0.4000E+01  0.3624E+00  0.3308E+02  0.4887E+02
 0.5000E+01  0.7074E-01  0.3110E+02  0.4829E+02
 0.6000E+01 -0.2272E+00  0.2635E+02  0.4340E+02
 0.7000E+01 -0.5048E+00  0.1924E+02  0.3463E+02
 0.8000E+01 -0.7374E+00  0.1042E+02  0.2277E+02
 0.9000E+01 -0.9041E+00  0.6594E+00  0.8875E+01
 0.1000E+02 -0.9900E+00 -0.9155E+01 -0.5813E+01
 0.1100E+02 -0.9875E+00 -0.1815E+02 -0.1998E+02
 0.1200E+02 -0.8968E+00 -0.2553E+02 -0.3237E+02
 0.1300E+02 -0.7259E+00 -0.3062E+02 -0.4186E+02
 0.1400E+02 -0.4903E+00 -0.3298E+02 -0.4761E+02
 0.1500E+02 -0.2108E+00 -0.3240E+02 -0.4911E+02
 0.1600E+02  0.8750E-01 -0.2892E+02 -0.4623E+02
 0.1700E+02  0.3780E+00 -0.2285E+02 -0.3921E+02
 0.1800E+02  0.6347E+00 -0.1475E+02 -0.2869E+02
 0.1900E+02  0.8347E+00 -0.5325E+01 -0.1561E+02
 0.2000E+02  0.9602E+00  0.4572E+01 -0.1135E+01
```

Figure 11.6 Results from Program 11.2 example

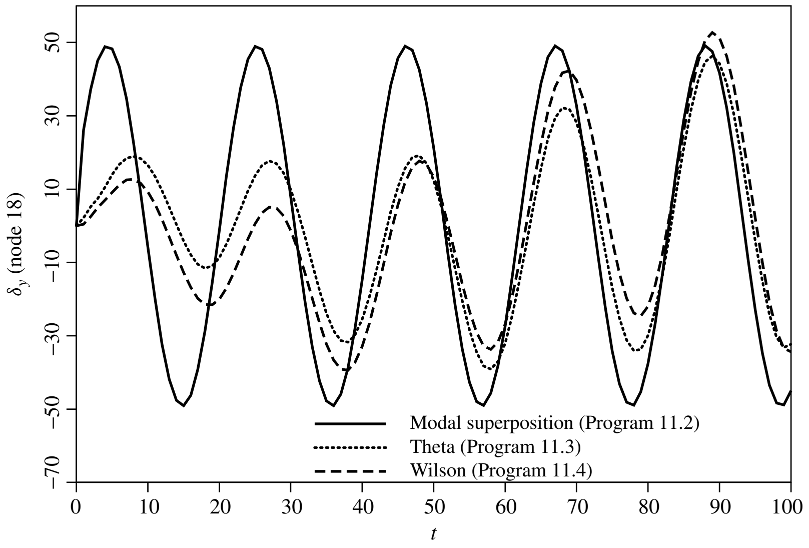


Figure 11.7 Cantilever tip displacement computed by Programs 11.2, 11.3 and 11.4

Program 11.3 Forced vibration analysis of an elastic solid in plane strain using rectangular 8-node quadrilaterals. Lumped or consistent mass. Mesh numbered in *x*- or *y*-direction. Implicit time integration using the "theta" method.

```
PROGRAM p113
!-------------------------------------------------------------------------
! Program 11.3 Forced vibration analysis of an elastic solid in plane
!              strain using rectangular 8-node quadrilaterals. Lumped or
!              consistent mass. Mesh numbered in x- or y-direction.
!              Implicit time integration using the "theta" method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=9,nn,nod=8,  &
   nodof=2,npri,nprops=3,np_types,nr,nres,nst=3,nstep,nxe,nye
 REAL(iwp)::area,c1,c2,c3,c4,det,dtim,fk,fm,one=1.0_iwp,theta,time,       &
   zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::consistent=.FALSE.
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   node(:),num(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),dee(:,:),der(:,:),deriv(:,:), &
   d1x0(:),d1x1(:),d2x0(:),d2x1(:),ecm(:,:),fun(:),f1(:),g_coord(:,:),    &
   jac(:,:),km(:,:),kv(:),loads(:),mm(:,:),mv(:),points(:,:),prop(:,:),   &
   val(:,:),weights(:),x0(:),x1(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),         &
   dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),&
   deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),num(nod),g_num(nod,nels),  &
   g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),fun(nod),etype(nels),      &
```

```
  prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres,fm,fk
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(x0(0:neq),d1x0(0:neq),x1(0:neq),d2x0(0:neq),loads(0:neq),      &
  d1x1(0:neq),d2x1(0:neq),kdiag(neq))
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
CALL sample(element,points,weights); kdiag=0
!-----------------------loop the elements to find global array sizes------
elements_1: DO iel=1,nels
  CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
  CALL num_to_g(num,nf,g); g_num(:,iel)=num
  g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),mv(kdiag(neq)),f1(kdiag(neq)))
WRITE(11,'(2(A,I5))')                                                   &
  " There are",neq," equations and the skyline storage is",kdiag(neq)
kv=zero; mv=zero
!-----------------------global stiffness and mass matrix assembly---------
elements_2: DO iel=1,nels
  CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
  num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
  g=g_g(:,iel); km=zero; mm=zero; area=zero
  gauss_pts_1: DO i=1,nip
    CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    area=area+det*weights(i)
    IF(consistent)THEN; CALL ecmat(ecm,fun,ndof,nodof)
      mm=mm+ecm*det*weights(i)*prop(3,etype(iel))
    END IF
  END DO gauss_pts_1
  IF(.NOT.consistent)CALL elmat(area,prop(3,etype(iel)),mm)
  CALL fsparv(kv,km,g,kdiag); CALL fsparv(mv,mm,g,kdiag)
END DO elements_2
!-----------------------initial conditions and factorise equations--------
x0=zero; d1x0=zero; d2x0=zero
c1=(one-theta)*dtim; c2=fk-c1; c3=fm+one/(theta*dtim); c4=fk+theta*dtim
f1=c3*mv+c4*kv; CALL sparin(f1,kdiag); time=zero
!-----------------------time stepping loop------------------------------
WRITE(11,'(/A,I5))')" Result at node",nres
WRITE(11,'(A)')"    time        load        x-disp      y-disp"
WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))
timesteps: DO j=1,nstep
  time=time+dtim; loads=zero; x1=c3*x0+d1x0/theta
  DO i=1,loaded_nodes
    loads(nf(:,node(i)))=                                               &
      val(i,:)*(theta*dtim*load(time)+c1*load(time-dtim))
  END DO; CALL linmul_sky(mv,x1,d1x1,kdiag); d1x1=loads+d1x1; loads=c2*x0
  CALL linmul_sky(kv,loads,x1,kdiag); x1=x1+d1x1; CALL spabac(f1,x1,kdiag)
  d1x1=(x1-x0)/(theta*dtim)-d1x0*(one-theta)/theta
  d2x1=(d1x1-d1x0)/(theta*dtim)-d2x0*(one-theta)/theta
  x0=x1; d1x0=d1x1; d2x0=d2x1
  IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))
END DO timesteps
```

```
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!-----------------------Load-time function--------------------------------
 IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 REAL(iwp),INTENT(IN)::t
 REAL(iwp)::load_result
 load_result=COS(0.3_iwp*t)
RETURN
END FUNCTION load
END PROGRAM p113
```

New scalar integers:

`loaded_nodes`	number of loaded nodes
`nip`	number of integrating points
`nst`	number of stress terms

New scalar reals:

`c1`	working constant
`c2`	working constant
`c3`	working constant
`c4`	working constant
`det`	determinant of Jacobian matrix
`pt2`	set to 0.2
`pt25`	set to 0.25
`theta`	time integration weighting parameter

Scalar logical:

`consistent`	set to `.TRUE.` if mass matrix is “consistent”

New Dynamic integer arrays:

`node`	nodes to be loaded

New Dynamic real arrays:

`bee`	strain-displacement matrix
`dee`	stress–strain matrix
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`d1x0`	“old” velocities
`d1x1`	“new” velocities
`d2x0`	“old” accelerations
`d2x1`	“new” accelerations
`ecm`	element consistent mass matrix
`fun`	shape functions
`f1`	left hand side matrix (stored as a skyline)
`jac`	Jacobian matrix
`loads`	nodal loads and displacements
`points`	integrating point local coordinates

`val`	applied nodal load weightings
`weights`	weighting coefficients
`x0`	"old" displacements
`x1`	"new" displacements

In this program whose structure chart is given as Figure 11.8, the same problem as that previously analysed is solved again using a direct time-integration procedure. In this example the tip loading takes the form of a continuous cosine function defined in a `FUNCTION` subprogram called `load` placed at the end of the main program. The specific method is described in Section 3.13.2 where it was shown to be the same implicit technique as was used for first order problems in Program 8.1, where it is often called the "Crank–Nicolson" approach ($\theta = 0.5$). In second-order problems, it is also known as the "Newmark $\beta = 1/4$" method in which form it was used in Program 11.1.

To step from one time instant to the next, a set of simultaneous equations has to be solved. Since the differential equations are often linearised, this is not as great a numerical task as might be supposed because the equation coefficients are constant, and need be factorised only once before the time-stepping procedure commences (see equation (3.139)). Velocities and accelerations are computed by ancillary equations (3.140) and (3.141).

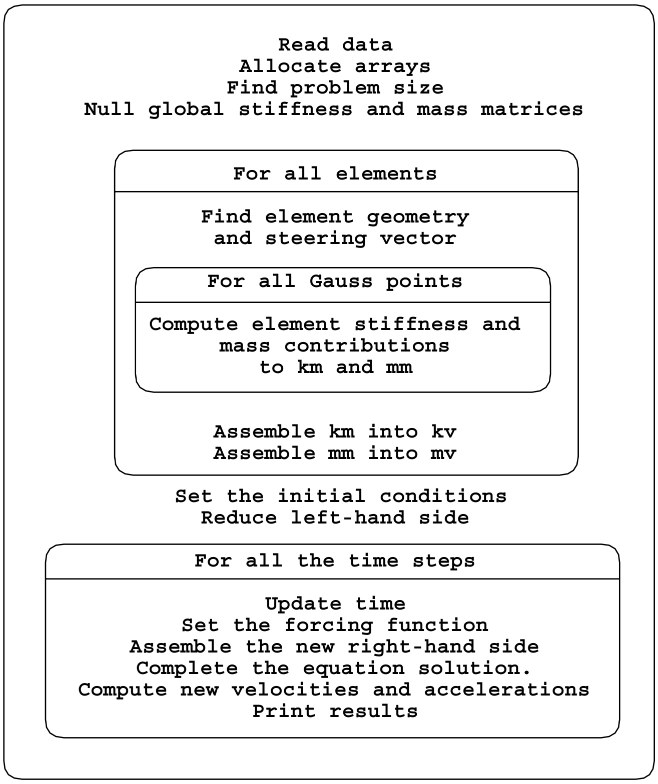
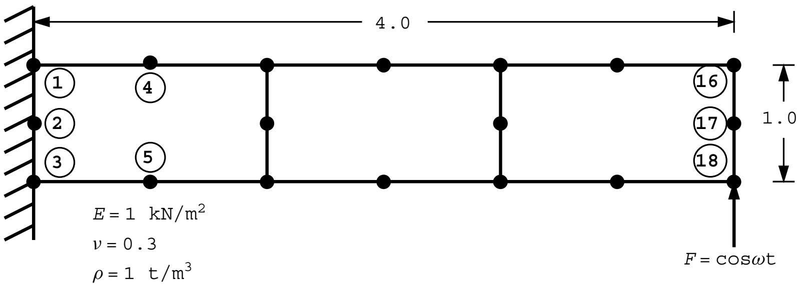

Read data
Allocate arrays
Find problem size
Null global stiffness and mass matrices
For all elements
Find element geometry and steering vector
For all Gauss points
Compute element stiffness and mass contributions to km and mm
Assemble km into kv
Assemble mm into mv
Set the initial conditions
Reduce left-hand side
For all the time steps
Update time
Set the forcing function
Assemble the new right-hand side
Complete the equation solution.
Compute new velocities and accelerations
Print results

Figure 11.8 Structure chart for implicit algorithms used in Programs 11.3 and 11.4

The example presented here uses lumped mass (`consistent = .FALSE.`), but the program can also handle consistent mass if required, in which case the element mass matrix is formed by subroutine `ecmat`. It should be noted that "exact" integration is needed to integrate the consistent mass matrix of a general 8-node quadrilateral element. In this example therefore, 9 integrating points (`nip=9`) have been used for all the element integrations. If, as is often the case, "reduced" (`nip=4`) integration is preferred in the generation of the stiffness matrix, two separate integration loops would be required, one for the (consistent) mass, and one for the stiffness.

Up to the section headed "global stiffness and mass matrix assembly" the program's task is the familiar one of generating the global stiffness and mass matrices, stored as usual as skyline vectors `kv` and `mv` respectively. The matrix arising on the left-hand side of (3.139) is then created, called (`f1`) and factorised using subroutine `sparin`.

In the time-stepping loop, the matrix-by-vector multiplications and vector additions specified on the right-hand side of (3.139) are carried out and equation solution is completed



```
nxe  nye
3     1

np_types
1

prop(e,v,rho)
1.0  0.3  1.0

etype (not needed)

x_coords, y_coords
0.0  1.33333  2.66667  4.0
0.0 -1.0

dtim  nstep  theta  npri  nres   fm       fk
1.0    20     0.5     1    18   0.005    0.272

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
1
18 0.0 1.0
```

Figure 11.9 Mesh and data for Program 11.3 example

```
There are   30 equations and the skyline storage is  285

Result at node   18
   time        load        x-disp      y-disp
 0.0000E+00  0.1000E+01  0.0000E+00  0.0000E+00
 0.1000E+01  0.9553E+00  0.7363E+00  0.2167E+01
 0.2000E+01  0.8253E+00  0.2231E+01  0.5226E+01
 0.3000E+01  0.6216E+00  0.3252E+01  0.7398E+01
 0.4000E+01  0.3624E+00  0.3987E+01  0.1046E+02
 0.5000E+01  0.7074E-01  0.4832E+01  0.1398E+02
 0.6000E+01 -0.2272E+00  0.5272E+01  0.1678E+02
 0.7000E+01 -0.5048E+00  0.5086E+01  0.1839E+02
 0.8000E+01 -0.7374E+00  0.4370E+01  0.1894E+02
 0.9000E+01 -0.9041E+00  0.3255E+01  0.1834E+02
 0.1000E+02 -0.9900E+00  0.1789E+01  0.1650E+02
 0.1100E+02 -0.9875E+00  0.8872E-01  0.1349E+02
 0.1200E+02 -0.8968E+00 -0.1674E+01  0.9574E+01
 0.1300E+02 -0.7259E+00 -0.3372E+01  0.5082E+01
 0.1400E+02 -0.4903E+00 -0.4915E+01  0.3726E+00
 0.1500E+02 -0.2108E+00 -0.6150E+01 -0.4128E+01
 0.1600E+02  0.8750E-01 -0.6858E+01 -0.7908E+01
 0.1700E+02  0.3780E+00 -0.6839E+01 -0.1050E+02
 0.1800E+02  0.6347E+00 -0.5986E+01 -0.1157E+02
 0.1900E+02  0.8347E+00 -0.4304E+01 -0.1096E+02
 0.2000E+02  0.9602E+00 -0.1902E+01 -0.8708E+01
```

Figure 11.10 Results from Program 11.3 example

by subroutine `spabac`. It then remains only to compute the new velocities and accelerations using (3.140) and (3.141).

The problem layout and data are given in Figure 11.9. Following the usual data relating to element numbers, properties and coordinates, the time stepping data calls for `nstep=20` steps of time step `dtim=1.0` with the output at node `nres=18` printed every `npri=1` time steps. The beam is damped using Rayleigh damping constants $f_m = 0.005$ and $f_k = 0.272$ (read as `fm` and `fk` respectively) giving a damping ratio ζ close to 0.05 (see equation (3.120)) for the first three natural frequencies. The results are listed as Figure 11.10 and plotted in Figure 11.7. The displacements are seen to be considerably disturbed by the initial condition “transients”, but once the response has settled down, the average amplitude agrees closely with that obtained by Modal Superposition, with a phase shift of about 90° relative to the loading function.

Program 11.4 Forced vibration analysis of an elastic solid in plane strain using rectangular 8-node quadrilaterals. Lumped or consistent mass. Mesh numbered in x- or y-direction. Implicit time integration using Wilson’s method.

```
PROGRAM p114
!-------------------------------------------------------------------------
! Program 11.4 Forced vibration analysis of an elastic solid in plane
!              strain using rectangular 8-node quadrilaterals. Lumped or
!              consistent mass. Mesh numbered in x- or y-direction.
!              Implicit time integration using Wilson's method.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=9,nn,nod=8,  &
   nodof=2,npri,nprops=3,np_types,nr,nres,nst=3,nstep,nxe,nye
 REAL(iwp)::area,c1,c2,c3,c4,det,dtim,d6=6.0_iwp,fk,fm,one=1.0_iwp,       &
   pt5=0.5_iwp,theta,time,two=2.0_iwp,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'; LOGICAL::consistent=.FALSE.
!-----------------------dynamic arrays-----------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag(:),nf(:,:), &
   node(:),num(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),dee(:,:),der(:,:),deriv(:,:), &
   d1x0(:),d1x1(:),d2x0(:),d2x1(:),ecm(:,:),fun(:),f1(:),g_coord(:,:),    &
   jac(:,:),km(:,:),kv(:),loads(:),mm(:,:),mv(:),points(:,:),prop(:,:),   &
   val(:,:),weights(:),x0(:),x1(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation-------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),         &
   dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),&
   deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),num(nod),g_num(nod,nels),  &
   g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),fun(nod),etype(nels),      &
   prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres,fm,fk
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(x0(0:neq),d1x0(0:neq),x1(0:neq),d2x0(0:neq),loads(0:neq),       &
   d1x1(0:neq),d2x1(0:neq),kdiag(neq))
 READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
 CALL sample(element,points,weights); kdiag=0
!-----------------------loop the elements to find global array sizes------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
 ALLOCATE(kv(kdiag(neq)),mv(kdiag(neq)),f1(kdiag(neq)))
 WRITE(11,'(2(A,I5))')                                                    &
   " There are",neq," equations and the skyline storage is",kdiag(neq)
 kv=zero; mv=zero
!-----------------------global stiffness and mass matrix assembly---------
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   g=g_g(:,iel); km=zero; mm=zero; area=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
     area=area+det*weights(i)
     IF(consistent)THEN; CALL ecmat(ecm,fun,ndof,nodof)
       mm=mm+ecm*det*weights(i)*prop(3,etype(iel))
     END IF
   END DO gauss_pts_1
   IF(.NOT.consistent)CALL elmat(area,prop(3,etype(iel)),mm)
   CALL fsparv(kv,km,g,kdiag); CALL fsparv(mv,mm,g,kdiag)
 END DO elements_2
!-----------------------initial conditions and factorise equations--------
 x0=zero; d1x0=zero; d2x0=zero
 c1=d6/(theta*dtim)**2; c2=c1*theta*dtim; c3=dtim**2/d6; c4=pt5*theta*dtim
 f1=(c1+pt5*c2*fm)*mv+(one+pt5*c2*fk)*kv; CALL sparin(f1,kdiag); time=zero
!-----------------------time stepping loop---------------------------------
 WRITE(11,'(/A,I5))')" Result at node",nres
 WRITE(11,'(A)')"    time        load        x-disp      y-disp"
 WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))
 timesteps: DO j=1,nstep
   time=time+dtim; loads=zero
   x1=(c1+pt5*c2*fm)*x0+(c2+two*fm)*d1x0+(two+c4*fm)*d2x0
   DO i=1,loaded_nodes
     loads(nf(:,node(i)))=                                                &
       val(i,:)*(theta*load(time)+(one-theta)*load(time-dtim))
   END DO; CALL linmul_sky(mv,x1,d1x1,kdiag); d1x1=loads+d1x1
   loads=pt5*c2*fk*x0+two*fk*d1x0+c4*fk*d2x0
   CALL linmul_sky(kv,loads,x1,kdiag); x1=x1+d1x1; CALL spabac(f1,x1,kdiag)
   d2x1=d2x0+((x1-x0)*c1-d1x0*c2-d2x0*two-d2x0)/theta
   d1x1=d1x0+pt5*dtim*(d2x1+d2x0); x0=x0+dtim*d1x0+two*c3*d2x0+d2x1*c3
   d1x0=d1x1; d2x0=d2x1
   IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))
 END DO timesteps
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!-----------------------Load-time function---------------------------------
 IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 REAL(iwp),INTENT(IN)::t
 REAL(iwp)::load_result
 load_result=COS(0.3_iwp*t)
RETURN
END FUNCTION load
END PROGRAM p114
```

New scalar reals:

`d6`	set to 6.0
`two`	set to 2.0

This algorithm is described in Section 3.13.3. The essential step is that shown in (3.142), which is of exactly the same form as (3.139) for the "theta" method, and so this program can be expected to resemble the previous one very closely. The structure chart of Figure 11.8 is again appropriate and the problem layout and data given in Figure 11.11 are essentially the same as in Figure 11.9. No new variables are involved, but the parameter θ is set to

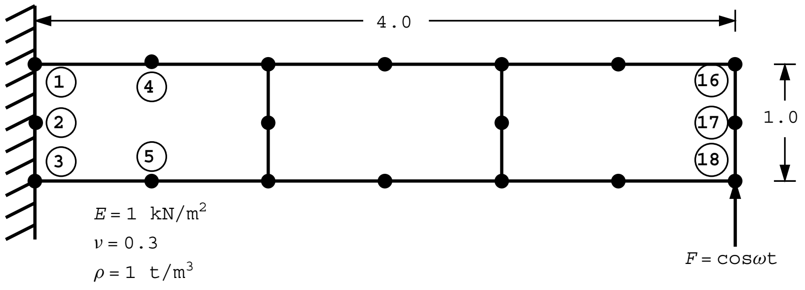


Figure 11.11 Mesh and data for Program 11.4 example (*Continued on page 486*)

```
nxe  nye
3     1

np_types
1

prop(e,v,rho)
1.0  0.3  1.0

etype (not needed)

x_coords, y_coords
0.0  1.33333  2.66667  4.0
0.0 -1.0

dtim  nstep  theta  npri  nres   fm      fk
1.0    20    1.4     1    18    0.005   0.272

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
1
18 0.0 1.0
```

Figure 11.11 (*Continued from page 485*)

```
There are   30 equations and the skyline storage is  285

Result at node   18
   time        load        x-disp      y-disp
 0.0000E+00  0.1000E+01  0.0000E+00  0.0000E+00
 0.1000E+01  0.9553E+00  0.1544E+00  0.4329E+00
 0.2000E+01  0.8253E+00  0.9763E+00  0.2575E+01
 0.3000E+01  0.6216E+00  0.2083E+01  0.4966E+01
 0.4000E+01  0.3624E+00  0.2908E+01  0.7004E+01
 0.5000E+01  0.7074E-01  0.3396E+01  0.9110E+01
 0.6000E+01 -0.2272E+00  0.3657E+01  0.1117E+02
 0.7000E+01 -0.5048E+00  0.3581E+01  0.1253E+02
 0.8000E+01 -0.7374E+00  0.3002E+01  0.1270E+02
 0.9000E+01 -0.9041E+00  0.1916E+01  0.1157E+02
 0.1000E+02 -0.9900E+00  0.4607E+00  0.9270E+01
 0.1100E+02 -0.9875E+00 -0.1204E+01  0.5928E+01
 0.1200E+02 -0.8968E+00 -0.2935E+01  0.1730E+01
 0.1300E+02 -0.7259E+00 -0.4592E+01 -0.3044E+01
 0.1400E+02 -0.4903E+00 -0.6033E+01 -0.8013E+01
 0.1500E+02 -0.2108E+00 -0.7122E+01 -0.1275E+02
 0.1600E+02  0.8750E-01 -0.7739E+01 -0.1682E+02
 0.1700E+02  0.3780E+00 -0.7769E+01 -0.1984E+02
 0.1800E+02  0.6347E+00 -0.7121E+01 -0.2149E+02
 0.1900E+02  0.8347E+00 -0.5759E+01 -0.2156E+02
 0.2000E+02  0.9602E+00 -0.3736E+01 -0.2002E+02
```

Figure 11.12 Results from Program 11.4 example

its stability limit of about 1.4, compared with 0.5 in the previous algorithm. All that need be said is that the `f1` matrix is now constructed as demanded by (3.142). The remaining steps of (3.144–3.147) are carried out within the section headed “time stepping loop”.

The results for one cycle are listed as Figure 11.12 and plotted in Figure 11.7. Again, the early stages of the response reflect mainly the influence of the start-up conditions. The response eventually settles down to be in good agreement with the previous solutions, although with a slightly greater phase shift.

Program 11.5 Forced vibration analysis of an elastic solid in plane strain using rectangular uniform size 4-node quadrilaterals. Mesh numbered in the *x*- or *y*-direction. Lumped or consistent mass. Mixed explicit/implicit time integration.

```
PROGRAM p115
!-------------------------------------------------------------------------
! Program 11.5 Forced vibration analysis of an elastic solid in plane
!              strain using rectangular uniform size 4-node quadrilaterals.
!              Mesh numbered in the x- or y-direction. Lumped and/or
!              consistent mass. Mixed explicit/implicit time integration.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,j,iel,k,ndim=2,ndof=8,nels,neq,nip=4,nn,nod=4,nodof=2,npri,   &
   nprops=3,np_types,nr,nres,nstep,nxe,nye
 REAL(iwp)::area,beta,c1,det,dtim,gamma,one=1.0_iwp,pt5=0.5_iwp,time,     &
   two=2.0_iwp,zero=0.0_iwp; CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),kdiag_l(:),       &
   kdiag_r(:),nf(:,:),num(:)
 REAL(iwp),ALLOCATABLE::coord(:,:),der(:,:),d1x0(:),d1x1(:),d2x0(:),      &
   d2x1(:),ecm(:,:),fun(:),g_coord(:,:),jac(:,:),km(:,:),kv(:),mm(:,:),   &
   mv(:),points(:,:),prop(:,:),weights(:),x0(:),x1(:),x_coords(:),        &
   y_coords(:); CHARACTER(LEN=1),ALLOCATABLE::mtype(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),         &
   coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),             &
   km(ndof,ndof),num(nod),g_num(nod,nels),g_g(ndof,nels),mtype(nels),     &
   mm(ndof,ndof),ecm(ndof,ndof),fun(nod),prop(nprops,np_types),           &
   x_coords(nxe+1),y_coords(nye+1),etype(nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,beta,gamma,npri,nres
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(kdiag_l(neq),kdiag_r(neq),x0(0:neq),d1x0(0:neq),x1(0:neq),      &
   d2x0(0:neq),d1x1(0:neq),d2x1(0:neq))
 READ(10,*)mtype; CALL sample(element,points,weights); kdiag_l=0; kdiag_r=0
!-----------------------loop the elements to find global array sizes------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
   CALL fkdiag(kdiag_r,g); IF(mtype(iel)=='c')CALL fkdiag(kdiag_l,g)
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 WHERE(kdiag_l==0); kdiag_l=1; END WHERE
 DO i=2,neq; kdiag_l(i)=kdiag_l(i)+kdiag_l(i-1)
   kdiag_r(i)=kdiag_r(i)+kdiag_r(i-1)
 END DO; ALLOCATE(kv(kdiag_l(neq)),mv(kdiag_r(neq)))
 WRITE(11,'(A,I5,A,/,2(A,I5),A)')                                        &
   " There are",neq," equations."," Skyline storage is",kdiag_l(neq),    &
   " to the left, and",kdiag_r(neq)," to the right."
 c1=one/dtim/dtim/beta; kv=zero; mv=zero
!-----------------------global stiffness and mass matrix assembly---------
 elements_2: DO iel=1,nels
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
   CALL stiff4(km,coord,prop(1,etype(iel)),prop(2,etype(iel)))
   g=g_g(:,iel); area=zero; mm=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); area=area+det*weights(i)
     CALL shape_fun(fun,points,i)
     IF(mtype(iel)=='c')THEN; CALL ecmat(ecm,fun,ndof,nodof)
       mm=mm+ecm*det*weights(i)*c1*prop(3,etype(iel))
     END IF
   END DO gauss_pts_1
   IF(mtype(iel)=='l')THEN
     CALL elmat(area,c1*prop(3,etype(iel)),mm)
     CALL fsparv(kv,mm,g,kdiag_l); CALL fsparv(mv,mm-km,g,kdiag_r); ELSE
     CALL fsparv(kv,km+mm,g,kdiag_l); CALL fsparv(mv,mm,g,kdiag_r)
   END IF
 END DO elements_2
!-----------------------initial conditions and factorise equations--------
 x0=zero; d1x0=one; d2x0=zero; CALL sparin(kv,kdiag_l); time=zero
!-----------------------time stepping loop-------------------------------
 WRITE(11,'(/A,I5))')" Result at node",nres
 WRITE(11,'(A)')"    time        x-disp      y-disp"
 WRITE(11,'(4E12.4)')time,x0(nf(:,nres))
 timesteps: DO j=1,nstep
   time=time+dtim; d1x1=x0+d1x0*dtim+d2x0*pt5*dtim*dtim*(one-two*beta)
   CALL linmul_sky(mv,d1x1,x1,kdiag_r); CALL spabac(kv,x1,kdiag_l)
   d2x1=(x1-d1x1)/dtim/dtim/beta
   d1x1=d1x0+d2x0*dtim*(one-gamma)+d2x1*dtim*gamma
   x0=x1; d1x0=d1x1; d2x0=d2x1
   IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,x0(nf(:,nres))
 END DO timesteps
STOP
END PROGRAM p115
```

New dynamic integer arrays:

`kdiag_l`	diagonal term locations on the left
`kdiag_r`	diagonal term locations on the right

New dynamic character array:

`mtype`	set to `'l'` for lumped mass and `'c'` for consistent

Programs 11.3 and 11.4 used implicit time-marching algorithms, and Program 11.7 described later in this chapter uses an explicit approach. The program described now combines the methods of implicit and explicit time integration in a single program. Although not "mesh free" this procedure should allow economical bandwidths of the assembled matrices. The idea is (e.g. Key, 1980) that a mesh may contain only a few elements which have a very small explicit stability limit and are therefore best integrated implicitly. The remainder of the mesh can be successfully integrated explicitly at reasonable time steps.

The recurrence relations (3.139–3.141) are cast in the form:

$$\left(\frac{1}{\Delta t^2\beta}[\mathbf{M}_m] + [\mathbf{K}_m]\right)\{\mathbf{U}\}_1 = \{\mathbf{F}\}_1 + \frac{1}{\Delta t^2\beta}[\mathbf{M}_m]\left\{\overline{\mathbf{U}}\right\}_1 \tag{11.4}$$

for implicit elements, and

$$\frac{1}{\Delta t^2\beta}[\mathbf{M}_m]\{\mathbf{U}\}_1 = \{\mathbf{F}\}_1 + \left(\frac{1}{\Delta t^2\beta}[\mathbf{M}_m] - [\mathbf{K}_m]\right)\left\{\overline{\mathbf{U}}\right\}_1 \tag{11.5}$$

for explicit elements, where

$$\left\{\overline{\mathbf{U}}\right\}_1 = \{\mathbf{U}\}_0 + \Delta t\{\dot{\mathbf{U}}\}_0 + \frac{\Delta t^2(1-2\beta)}{2}\{\ddot{\mathbf{U}}\}_0 \tag{11.6}$$

Accelerations and velocities are obtained from:

$$\{\ddot{\mathbf{U}}\}_1 = \frac{1}{\Delta t^2\beta}\left(\{\mathbf{U}\}_1 - \left\{\overline{\mathbf{U}}\right\}_1\right) \tag{11.7}$$

and

$$\{\dot{\mathbf{U}}\}_1 = \{\dot{\mathbf{U}}\}_0 + \Delta t(1-\gamma)\{\ddot{\mathbf{U}}\}_0 + \Delta t\gamma\{\ddot{\mathbf{U}}\}_1 \tag{11.8}$$

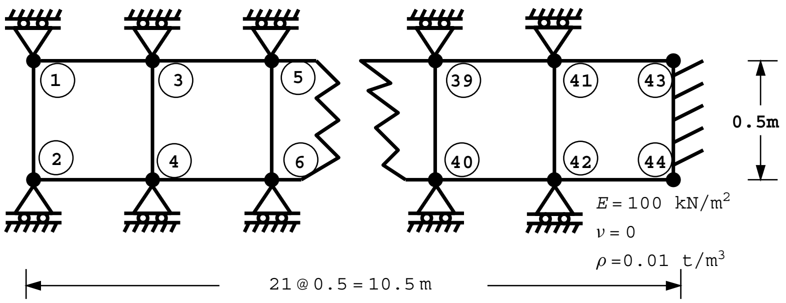
The time integration parameters are the "Newmark" ones introduced in Program 11.1 and conventionally called $\beta = 1/4$ and $\gamma = 1/2$ corresponding to $\theta = 1/2$ in Program 11.3.

When an explicit element is not coupled to an implicit one, the half-bandwidth (excluding the diagonal) of the assembled equation coefficient matrix will only be 1, whereas the full half-bandwidth will apply for implicit elements. Full advantage can be taken of variable bandwidth or "skyline" storage on both sides of the equation in this case (see Smith, 1984). The problem chosen is illustrated in Figure 11.13 and models the impact of an elastic rod, initially travelling at a uniform unit velocity, with a rigid wall. The elastic rod is constrained to vibrate in the axial direction only, and is fixed at the right-hand end. Initial conditions of a uniform unit velocity are applied to all freedoms in the mesh at time $t = 0$. The appropriate structure chart is Figure 11.8 for implicit integration.

No damping is considered in this case, and the data calls for `nstep=400` calculation steps at a time step of `dtim=0.0025` to be performed. Output is requested every `npri=1` time steps at node `nres=42`, which is close to the impacted end of the rod. The boundary conditions place rollers on the top and bottom surfaces of the rod, although there would be no tendency for "bulging" in this case since Poisson's ratio is read as zero.

The only other new parameter is a dynamic character array `mtype` which is assigned from data, and holds for each element, the character string `'c'` or `'l'`, corresponding to consistent or lumped mass respectively. In the present example, elements 1, 11, and 21 have consistent mass while the others are lumped. Also, the lumped elements are explicitly integrated while the consistent ones are implicitly integrated.

The stiffness and mass are integrated as usual. When the element mass matrix is consistent (`mtype='c'`), (`mm+km`) is assembled into `kv` while the `mm` is assembled into `mv` as shown in (11.4) for implicit elements. Conversely, when the element mass matrix is lumped (`mtype= 'l'`), `mm` is assembled into `kv` while (`mm-km`) is assembled into `mv` as shown in (11.5) for explicit elements. The integer vectors `kdiag_l` and `kdiag_r` locate the diagonal terms of the left and right hand side matrices `kv` and `mv` respectively.



```
nxe  nye  np_types
21    1     1

prop(e,v,rho)
100.0  0.0  0.01

etype (not needed)

x_coords, y_coords
0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0 10.5
0.0 -0.5

dtim    nstep  beta     gamma  npri  nres
0.0025   400   0.25      0.5    1     42

nr,(k,nf(:,k),i=1,nr)
44
 1 1 0   2 1 0   3 1 0   4 1 0   5 1 0
 6 1 0   7 1 0   8 1 0   9 1 0  10 1 0
11 1 0  12 1 0  13 1 0  14 1 0  15 1 0
16 1 0  17 1 0  18 1 0  19 1 0  20 1 0
21 1 0  22 1 0  23 1 0  24 1 0  25 1 0
26 1 0  27 1 0  28 1 0  29 1 0  30 1 0
31 1 0  32 1 0  33 1 0  34 1 0  35 1 0
36 1 0  37 1 0  38 1 0  39 1 0  40 1 0
41 1 0  42 1 0  43 0 0  44 0 0

mtype
'c' 'l' 'l' 'l' 'l' 'l' 'l' 'l' 'l' 'l'
'c' 'l' 'l' 'l' 'l' 'l' 'l' 'l' 'l' 'l' 'c'
```

Figure 11.13 Mesh and data for Program 11.5 example

The initial conditions are then set, with the starting velocity in the x-direction at all nodes set to unity (`d1x0=1.0`). The global matrix factorisation is done by `sparin` and the time-stepping loop is entered. Equations (11.4) and (11.5) require the usual matrix-by-vector multiplication on the right hand side by subroutine `linmul_sky`. Equation solution is completed by `spabac` and it remains only to update velocities and accelerations for the next time step from (11.7–11.8).

The results are listed in Figure 11.14 and the displacements close to the support (freedom 42) are compared graphically with the exact solution in Figure 11.15. Despite some spurious oscillations the response is reasonably modelled.

```
 There are   42 equations.
 Skyline storage is   55 to the left, and  143 to the right.

 Result at node   42
    time         x-disp      y-disp
  0.0000E+00  0.0000E+00  0.0000E+00
  0.2500E-02  0.2326E-02  0.0000E+00
  0.5000E-02  0.4051E-02  0.0000E+00
  0.7500E-02  0.4898E-02  0.0000E+00
  0.1000E-01  0.5020E-02  0.0000E+00
  0.1250E-01  0.4834E-02  0.0000E+00
  0.1500E-01  0.4724E-02  0.0000E+00
  0.1750E-01  0.4825E-02  0.0000E+00
  0.2000E-01  0.5029E-02  0.0000E+00
  0.2250E-01  0.5150E-02  0.0000E+00
  0.2500E-01  0.5106E-02  0.0000E+00
.
.
.
  0.9775E+00  0.4544E-02  0.0000E+00
  0.9800E+00  0.4640E-02  0.0000E+00
  0.9825E+00  0.4821E-02  0.0000E+00
  0.9850E+00  0.5040E-02  0.0000E+00
  0.9875E+00  0.5250E-02  0.0000E+00
  0.9900E+00  0.5399E-02  0.0000E+00
  0.9925E+00  0.5420E-02  0.0000E+00
  0.9950E+00  0.5270E-02  0.0000E+00
  0.9975E+00  0.4973E-02  0.0000E+00
  0.1000E+01  0.4648E-02  0.0000E+00
```

Figure 11.14 Results from Program 11.5 example

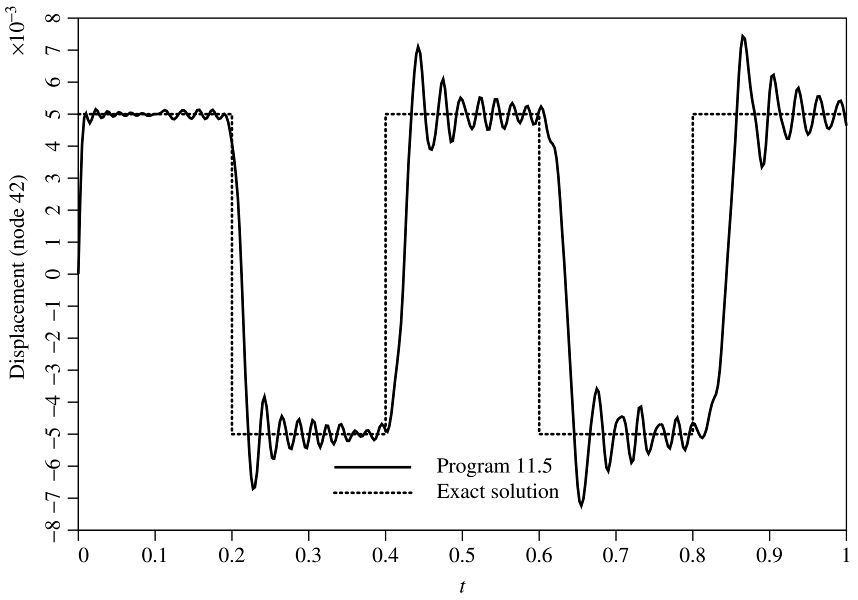


Figure 11.15 Displacement near support versus time from Program 11.5 example

Program 11.6 Forced vibration analysis of an elastic solid in plane strain using rectangular 8-node quadrilaterals. Lumped or consistent mass. Mesh numbered in *x*- or *y*-direction. Implicit time integration using the "theta" method. No global matrix assembly. Diagonally preconditioned conjugate gradient solver.

```
PROGRAM p116
!-------------------------------------------------------------------------
! Program 11.6 Forced vibration analysis of an elastic solid in plane
!              strain using rectangular 8-node quadrilaterals. Lumped or
!              consistent mass. Mesh numbered in x- or y-direction.
!              Implicit time integration using the "theta" method.
!              No global matrix assembly. Diagonally preconditioned
!              conjugate gradient solver.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::cg_iters,cg_limit,i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels,   &
   neq,nip=9,nn,nod=8,nodof=2,npri,nprops=3,np_types,nr,nres,nst=3,nstep, &
   nxe,nye
 REAL(iwp)::alpha,area,beta,cg_tol,c1,c2,c3,c4,det,dtim,fk,fm,one=1.0_iwp,&
   theta,time,up,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
 LOGICAL::consistent=.FALSE.,cg_converged
!-----------------------dynamic arrays------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),nf(:,:),node(:),  &
   num(:)
 REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),d(:),dee(:,:),der(:,:),       &
   deriv(:,:),diag_precon(:),d1x0(:),d1x1(:),d2x0(:),d2x1(:),ecm(:,:),    &
   fun(:),g_coord(:,:),jac(:,:),km(:,:),loads(:),mm(:,:),p(:),points(:,:),&
   prop(:,:),storkm(:,:,:),stormm(:,:,:),u(:),val(:,:),weights(:),x(:),   &
   xnew(:),x0(:),x1(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,cg_tol,cg_limit,np_types
 CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),         &
   dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),&
   deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),num(nod),g_num(nod,nels),  &
   g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),fun(nod),etype(nels),      &
   prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1),                 &
   storkm(ndof,ndof,nels),stormm(ndof,ndof,nels))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres,fm,fk
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(x0(0:neq),d1x0(0:neq),x1(0:neq),d2x0(0:neq),loads(0:neq),       &
   d1x1(0:neq),d2x1(0:neq),d(0:neq),p(0:neq),x(0:neq),xnew(0:neq),        &
   diag_precon(0:neq),u(0:neq))
 READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
!---------------loop the elements to set up element data------------------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1
 CALL sample(element,points,weights); diag_precon=zero
 WRITE(11,'(A,I5,A)')" There are",neq," equations"
 c1=(one-theta)*dtim; c2=fk-c1; c3=fm+one/(theta*dtim); c4=fk+theta*dtim
 CALL sample(element,points,weights); diag_precon=zero
!----element stiffness and mass integration, storage and preconditioner---
 elements_2: DO iel=1,nels
   CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
   num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
   km=zero; mm=zero; area=zero
   gauss_pts_1: DO i=1,nip
     CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
     jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
     deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
     km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
     area=area+det*weights(i)
     IF(consistent)THEN; CALL ecmat(ecm,fun,ndof,nodof)
       ecm=ecm*det*weights(i); mm=mm+ecm
     END IF
   END DO gauss_pts_1
   IF(.NOT.consistent)CALL elmat(area,prop(3,etype(iel)),mm)
   storkm(:,:,iel)=km; stormm(:,:,iel)=mm
   DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+mm(k,k)*c3+km(k,k)*c4
   END DO
 END DO elements_2
 diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
!----------------------time stepping loop-------------------------------
 x0=zero; d1x0=zero; d2x0=zero; time=zero
 WRITE(11,'(/A,I5))')" Result at node",nres
 WRITE(11,'(A)')                                                       &
   "    time        load        x-disp      y-disp     cg iters"
 WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))
 timesteps: DO j=1,nstep
   time=time+dtim; loads=zero; u=zero
   elements_3: DO iel=1,nels
     g=g_g(:,iel); km=storkm(:,:,iel); mm=stormm(:,:,iel)
     u(g)=u(g)+MATMUL(km*c2+mm*c3,x0(g))+MATMUL(mm/theta,d1x0(g))
   END DO elements_3; u(0)=zero
   DO i=1,loaded_nodes
     loads(nf(:,node(i)))=                                             &
       val(i,:)*(theta*dtim*load(time)+c1*load(time-dtim))
   END DO
   loads=u+loads; d=diag_precon*loads; p=d; x=zero; cg_iters=0
!----------------------pcg equation solution----------------------------
   pcg: DO
     cg_iters=cg_iters+1; u=zero
     elements_4: DO iel=1,nels
       g=g_g(:,iel); km=storkm(:,:,iel); mm=stormm(:,:,iel)
       u(g)=u(g)+MATMUL(mm*c3+km*c4,p(g))
     END DO elements_4; u(0)=zero
     up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
     loads=loads-u*alpha; d=diag_precon*loads; beta=DOT_PRODUCT(loads,d)/up
     p=d+p*beta; call checon(xnew,x,cg_tol,cg_converged)
     IF(cg_converged.OR.cg_iters==cg_limit)EXIT
   END DO pcg
   x1=xnew; d1x1=(x1-x0)/(theta*dtim)-d1x0*(one-theta)/theta
   d2x1=(d1x1-d1x0)/(theta*dtim)-d2x0*(one-theta)/theta
   IF(j/npri*npri==j)                                                  &
     WRITE(11,'(4E12.4,I8)')time,load(time),x1(nf(:,nres)),cg_iters
   x0=x1; d1x0=d1x1; d2x0=d2x1
 END DO timesteps
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!----------------------Load-time function-------------------------------
 IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 REAL(iwp),INTENT(IN)::t
 REAL(iwp)::load_result
 load_result=COS(0.3_iwp*t)
RETURN
END FUNCTION load
END PROGRAM p116
```

New scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling

New scalar reals:

`alpha`	α from equations (3.22)
`beta`	β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`up`	holds dot product $\{\mathbf{R}\}_k^{\mathrm{T}}\{\mathbf{R}\}_k$

New scalar logical:

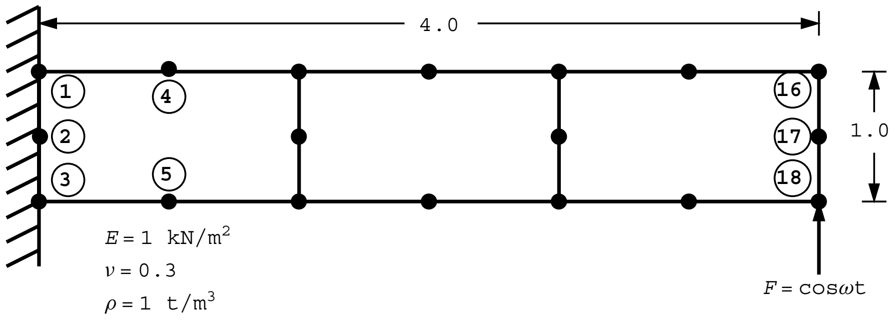
`cg_converged`	set to `.TRUE.` if pcg process has converged

New dynamic real arrays:

`d`	vector used in equation (3.22)
`diag_precon`	diagonal preconditioner vector
`p`	"descent" vector used in equations (3.22)
`storkm`	holds element stiffness matrices
`stormm`	holds element mass matrices
`u`	vector used in equation (3.22)
`x`	"old" solution vector
`xnew`	"new" solution vector

Typical mesh free strategies can preserve the unconditional stability of the "implicit" procedures such as in Program 11.3 by replacing the direct equation solution by an iterative approach such as pcg. Alternatively, as was done in Chapter 8 (Program 8.4), a purely explicit time-integration procedure for dynamic analysis can be adopted with its inherent stability limitations, as will be explained in the final Program 11.7. Product EBE techniques are also available (Wong *et al.*, 1989) but are not dealt with in this book.

Program 11.6 is adapted from Program 11.3 (implicit integration by the "theta" method) in the same way as Program 8.3 was adapted from Program 8.2. Equation solution is accomplished on every timestep by preconditioned conjugate gradients using diagonal preconditioning of the left hand side matrix in equation (3.139).



```
nxe  nye   cg_tol   cg_limit
3     1    1.0e-5    50

np_types
1

prop(e,v,rho)
1.0  0.3  1.0

etype (not needed)

x_coords, y_coords
0.0  1.33333  2.66667  4.0
0.0 -1.0

dtim  nstep  theta  npri  nres   fm      fk
1.0    20     0.5    1     18   0.005   0.272

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
1
18 0.0 1.0
```

Figure 11.16 Mesh and data for Program 11.6 example

The element stiffness matrices km are stored as storkm with the mass matrices mm as stormm. The same example analysed by Programs 11.2, 11.3, and 11.4 is repeated, with the data given in Figure 11.16. The results shown in Figure 11.17 are essentially the same as those obtained using a direct solver shown in Figure 11.10. It may also be noted from these results that this version of pcg solution is taking approximately neq/2 iterations to converge (where neq is the number of equations), a similar convergence rate as was usual in Chapters 5 and 6. In Chapter 8, the convergence rate was much quicker at approximately neq/10. Fortunately, as problem sizes increase the iteration count, as a proportion of neq, drops very rapidly.

```
 There are   30 equations

 Result at node   18
    time        load        x-disp      y-disp     cg iters
  0.0000E+00  0.1000E+01  0.0000E+00  0.0000E+00
  0.1000E+01  0.9553E+00  0.7363E+00  0.2167E+01      17
  0.2000E+01  0.8253E+00  0.2231E+01  0.5226E+01      17
  0.3000E+01  0.6216E+00  0.3252E+01  0.7398E+01      17
  0.4000E+01  0.3624E+00  0.3987E+01  0.1046E+02      17
  0.5000E+01  0.7074E-01  0.4832E+01  0.1398E+02      16
  0.6000E+01 -0.2272E+00  0.5272E+01  0.1678E+02      16
  0.7000E+01 -0.5048E+00  0.5086E+01  0.1839E+02      16
  0.8000E+01 -0.7374E+00  0.4370E+01  0.1894E+02      17
  0.9000E+01 -0.9041E+00  0.3255E+01  0.1834E+02      17
  0.1000E+02 -0.9900E+00  0.1789E+01  0.1650E+02      17
  0.1100E+02 -0.9875E+00  0.8842E-01  0.1349E+02      18
  0.1200E+02 -0.8968E+00 -0.1674E+01  0.9574E+01      18
  0.1300E+02 -0.7259E+00 -0.3373E+01  0.5082E+01      18
  0.1400E+02 -0.4903E+00 -0.4916E+01  0.3731E+00      17
  0.1500E+02 -0.2108E+00 -0.6151E+01 -0.4127E+01      17
  0.1600E+02  0.8750E-01 -0.6859E+01 -0.7907E+01      17
  0.1700E+02  0.3780E+00 -0.6839E+01 -0.1050E+02      17
  0.1800E+02  0.6347E+00 -0.5986E+01 -0.1157E+02      18
  0.1900E+02  0.8347E+00 -0.4304E+01 -0.1096E+02      18
  0.2000E+02  0.9602E+00 -0.1902E+01 -0.8706E+01      18
```

Figure 11.17 Results from Program 11.6 example

Program 11.7 Forced vibration analysis of an elastic–plastic (von Mises) solid in plane strain using rectangular 8-node quadrilateral elements. Lumped mass. Mesh numbered in x- or y-direction. Explicit time integration.

```
PROGRAM p117
!-------------------------------------------------------------------------
! Program 11.7 Forced vibration analysis of an elastic-plastic (Von Mises)
!              solid in plane strain using rectangular 8-node quadrilateral
!              elements. Lumped mass. Mesh numbered in x- or y-direction.
!              Explicit time integration.
!-------------------------------------------------------------------------
 USE main; USE geom; IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 INTEGER::i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nn,nod=8,  &
   nodof=2,npri,nprops=4,np_types,nr,nres,nst=4,nstep,nxe,nye
 REAL(iwp)::area,det,dsbar,dtim,f,fac,fmax,fnew,lode_theta,one=1.0_iwp,   &
   pt5=0.5_iwp,sigm,time,zero=0.0_iwp
 CHARACTER(LEN=15)::element='quadrilateral'
!-----------------------dynamic arrays-------------------------------------
 INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:),g_num(:,:),nf(:,:),node(:),  &
   num(:)
 REAL(iwp),ALLOCATABLE::bdylds(:),bee(:,:),bload(:),coord(:,:),dee(:,:),  &
   der(:,:),deriv(:,:),d1x1(:),d2x1(:),eld(:),eload(:),eps(:),            &
   etensor(:,:,:),diag(:),g_coord(:,:),jac(:,:),mm(:,:),pl(:,:),          &
   points(:,:),prop(:,:),sigma(:),stress(:),tensor(:,:,:),val(:,:),       &
   weights(:),x1(:),x_coords(:),y_coords(:)
!-----------------------input and initialisation--------------------------
 OPEN(10,FILE='fe95.dat'); OPEN(11,FILE='fe95.res')
 READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
 ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
   num(nod),dee(nst,nst),tensor(nst,nip,nels),coord(nod,ndim),pl(nst,nst),&
   etensor(nst,nip,nels),jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),    &
   g_num(nod,nels),bee(nst,ndof),eld(ndof),eps(nst),sigma(nst),           &
   mm(ndof,ndof),bload(ndof),eload(ndof),g(ndof),stress(nst),etype(nels), &
   g_g(ndof,nels),x_coords(nxe+1),y_coords(nye+1),prop(nprops,np_types))
 READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
 READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,npri,nres
 nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
 ALLOCATE(bdylds(0:neq),x1(0:neq),d1x1(0:neq),d2x1(0:neq),diag(0:neq))
 READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
 READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
!-----------------------loop the elements to set up global geometry ------
 elements_1: DO iel=1,nels
   CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
   CALL num_to_g(num,nf,g); g_num(:,iel)=num
   g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
 END DO elements_1; CALL mesh(g_coord,g_num,12)
 WRITE(11,'(A,I5,A)')"There are",neq," equations"
!-----------------------initial conditions--------------------------------
 tensor=zero; etensor=zero; x1=zero; d1x1=zero; d2x1=zero; diag=zero
 CALL sample(element,points,weights); time=zero
!-----------------------time stepping loop--------------------------------
 WRITE(11,'(/A,I5))')" Result at node",nres
 WRITE(11,'(A)')"    time        load        x-disp      y-disp"
 WRITE(11,'(4E12.4)')time,load(time),x1(nf(:,nres))
 time_steps: DO j=1,nstep
   fmax=zero; time=time+dtim; x1=x1+dtim*d1x1+pt5*dtim**2*d2x1; bdylds=zero
!-----------------------go round the Gauss Points ------------------------
   elements_2: DO iel=1,nels
     num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
     area=zero; bload=zero; eld=x1(g)
     gauss_pts_1: DO i=1,nip
       CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
       CALL shape_der(der,points,i); jac=MATMUL(der,coord)
       det=determinant(jac); area=area+det*weights(i); CALL invert(jac)
       deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
       eps=MATMUL(bee,eld); eps=eps-etensor(:,i,iel); sigma=MATMUL(dee,eps)
       stress=sigma+tensor(:,i,iel)
       CALL invar(stress,sigm,dsbar,lode_theta)
       fnew=dsbar-prop(4,etype(iel))
!-----------------------check whether yield is violated-------------------
       IF(fnew>=zero)THEN
         stress=tensor(:,i,iel); CALL invar(stress,sigm,dsbar,lode_theta)
         f=dsbar-prop(4,etype(iel)); fac=fnew/(fnew-f)
         stress=tensor(:,i,iel)+(one-fac)*sigma; CALL vmdpl(dee,stress,pl)
         dee=dee-fac*pl
       END IF
       sigma=MATMUL(dee,eps); sigma=sigma+tensor(:,i,iel)
       CALL invar(sigma,sigm,dsbar,lode_theta); f=dsbar-prop(4,etype(iel))
       IF(f>fmax)fmax=f; eload=MATMUL(sigma,bee)
       bload=bload+eload*det*weights(i)
!-----------------------update the Gauss Point stresses and strains-------
       tensor(:,i,iel)=sigma; etensor(:,i,iel)=etensor(:,i,iel)+eps
     END DO gauss_pts_1
     bdylds(g)=bdylds(g)-bload
     IF(j==1)THEN
       CALL elmat(area,prop(3,etype(iel)),mm); CALL formlump(diag,mm,g)
     END IF
   END DO elements_2; bdylds(0)=zero
   DO i=1,loaded_nodes
     bdylds(nf(:,node(i)))=bdylds(nf(:,node(i)))+val(i,:)*load(time)
   END DO
   bdylds(1:)=bdylds(1:)/diag(1:); d1x1=d1x1+(d2x1+bdylds)*pt5*dtim
   d2x1=bdylds; WRITE(*,'(A,I6,A,F8.4)')"  time step",j,"    F_max",fmax
   IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,load(time),x1(nf(:,nres))
 END DO time_steps
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!----------------------Load-time function-------------------------------
 IMPLICIT NONE
 INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
 REAL(iwp),INTENT(IN)::t
 REAL(iwp)::load_result
 load_result=-180.0_iwp
RETURN
END FUNCTION load
END PROGRAM p117
```

New scalar reals:

`dsbar`	invariant, $\bar{\sigma}$
`f`	value of yield function
`fac`	measure of yield surface overshoot (f from equation (6.35))
`fmax`	maximum value of yield function f at each time step
`fnew`	value of yield function after stress increment
`lode_theta`	Lode angle, θ
`sigm`	mean stress, σ_m

New dynamic real arrays:

`bdylds`	self-equilibrating global body forces
`bload`	self-equilibrating element body forces
`eld`	element nodal displacements
`eload`	integrating point contribution to `bload`
`eps`	strain terms
`etensor`	holds running total of all integrating point strain terms
`diag`	global lumped mass vector
`pl`	plastic $[\mathbf{D}^p]$ matrix (6.29)
`sigma`	stress terms
`stress`	stress term increments
`tensor`	holds running total of all integrating point stress terms

In the same way as was done for first order problems in Program 8.4, θ can be set to zero in second order recurrence formulae such as (3.139). After rearrangement, the only matrix remaining on the left hand side of the equation is $[\mathbf{M}_m]$. If this is lumped (diagonalised), the new solution $\{\mathbf{U}\}_1$ can be computed without solving simultaneous equations at all. Further, the right hand side products can again be completed using element-by-element summation and so no global matrices are involved. This procedure is particularly attractive in non-linear problems where the element stiffness $[\mathbf{k}_m]$ is a function of, for example, strain. In the present program, non-linearity is introduced in the form of elasto-plasticity, which

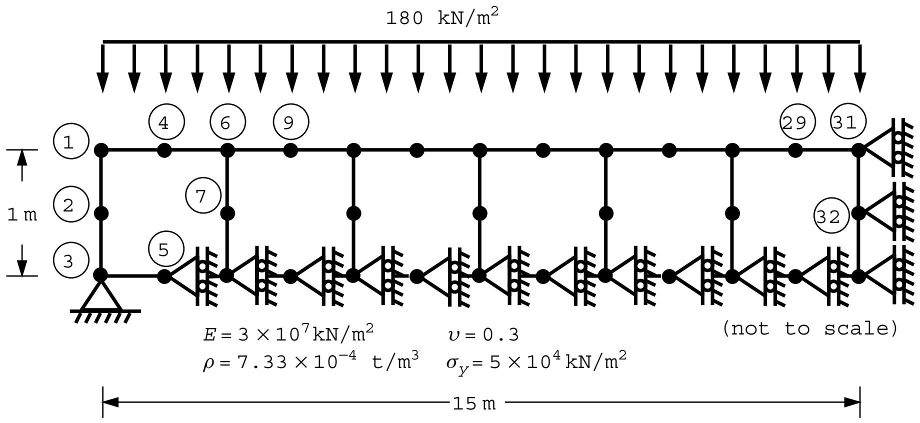
```
Initialisation, e.g.
$\{U_0\} = \{\dot{U}_0\} = \{\ddot{U}_0\} = \{0\}$
    For all time steps
        Replace $\{U_1\}$ by $\{U_0\} + \Delta t\{\dot{U}_0\} + 0.5\Delta t\{\ddot{U}_0\}$
        For all elements
            Retrieve element geometry.
            Find nodal displacments $\{u\}$.
            For all Gauss points
                Find the [B] and [D] matrices.
                Find the element strains $\{\varepsilon\} = [B]\{u\}$
                Find the element stresses $\{\sigma\} = [D]\{\varepsilon\}$
                Find the element internal forces
                $\{bload\} = \int [B]^T\{\sigma\}dV$
            Assemble global internal forces
            $\{bdylds\}$
        Add external loads to internal loads
        Compute $\{\ddot{U}_1\} = [M]^{-1} * \{bdylds\}$
        Replace $\{\dot{U}_1\}$ by $\{\dot{U}_0\} + 0.5\Delta t(\{\ddot{U}_0\} + \{\ddot{U}_1\})$.
        Replace $\{\ddot{U}_1\}$ by $\{\ddot{U}_0\}$
```

Figure 11.18 Structure chart for Program 11.7

was described in Chapter 6. The nomenclature used is therefore drawn from the earlier programs in this chapter and from those in Chapter 6, particularly Program 6.1 which dealt with von Mises solids. The structure chart for the program is shown in Figure 11.18 and the problem layout and data in Figure 11.19.

Turning to the program code, after input and initialisation, the von Mises plastic stress–strain matrix $[\mathbf{D}^p]$ (called `pl` in the program, see Appendix C) is formed by subroutine `vmdpl`. The remainder of the program is a large explicit integration time-stepping loop. The displacements (called `x1`) are updated and then, scanning all elements and Gauss points, new strains can be computed. The constitutive relation then determines the appropriate level of stress and hence whether the yield stress has been violated or not. If the yield stress has not been violated, the material remains elastic, otherwise the constitutive matrix is updated by subtracting a proportion of the plastic matrix `pl` from the elastic matrix `dee` (see Figure 6.7). The corrected stresses are then redistributed as "body loads" `bdylds`, whence the new accelerations (`d2x1`) can be found and integrated to find the new velocities (`d1x1`). The next cycle of displacements can then be updated. The load weightings are read as data, and the loading function is held in function subprogram called `load`.

The example problem shown in Figure 11.19, is of a simply supported plane strain slab subjected to a sudden application of a uniformly distributed load of 180 kN/m^2 which



```
nxe  nye
6     1

np_types
1

prop(e,v,rho,sigma_y)
3.0e7  0.3  7.33e-4  5.0e4

etype (not needed)

x_coords, y_coords
 0.0  2.5  5.0  7.5 10.0  12.5  15.0
 0.0 -1.0

dtim    nstep  npri nres
1.0e-6  10000   50   31

nr,(k,nf(:,k),i=1,nr)
15
 3 0 0   5 0 1   8 0 1  10 0 1  13 0 1  15 0 1  18 0 1  20 0 1
23 0 1  25 0 1  28 0 1  30 0 1  31 0 1  32 0 1  33 0 1

loaded_nodes(node(i),val(i,:),i=1,loaded_nodes)
13
 1  0.0  0.4167     4  0.0  1.6667    6  0.0  0.8333
 9  0.0  1.6667    11  0.0  0.8333   14  0.0  1.6667
16  0.0  0.8333    19  0.0  1.6667   21  0.0  0.8333
24  0.0  1.6667    26  0.0  0.8333   29  0.0  1.6667
31  0.0  0.4167
```

Figure 11.19 Mesh and data for Program 11.7 example

remains constant with time. Symmetry has been assumed at the centre of the beam, and along the neutral axis, where only vertical movement is permitted. There are four properties required (`nprops=4`) in this non-linear analysis, namely Young's modulus E, Poisson's ratio ν, the mass density ρ and the (von Mises) yield strength of the material σ_y. The data calls for `nstep=10000` calculation time steps of length `dtim=1.0e-6`. Results are to be printed every `npri=50` steps at node `nres=31` which lies on the centreline of the beam.

A truncated set of results from the program is printed in Figure 11.20 in the form of elapsed time, load and the x- and y-displacements at node 31. Figure 11.21 gives a plot of

```
There are   50 equations

 Result at node   31
    time        load        x-disp       y-disp
  0.0000E+00 -0.1800E+03  0.0000E+00   0.0000E+00
  0.5000E-04 -0.1800E+03  0.0000E+00  -0.2995E-03
  0.1000E-03 -0.1800E+03  0.0000E+00  -0.1214E-02
  0.1500E-03 -0.1800E+03  0.0000E+00  -0.2684E-02
  0.2000E-03 -0.1800E+03  0.0000E+00  -0.4867E-02
  0.2500E-03 -0.1800E+03  0.0000E+00  -0.8084E-02
  0.3000E-03 -0.1800E+03  0.0000E+00  -0.1231E-01
  0.3500E-03 -0.1800E+03  0.0000E+00  -0.1742E-01
  0.4000E-03 -0.1800E+03  0.0000E+00  -0.2331E-01
  0.4500E-03 -0.1800E+03  0.0000E+00  -0.2991E-01
  0.5000E-03 -0.1800E+03  0.0000E+00  -0.3724E-01
  0.5500E-03 -0.1800E+03  0.0000E+00  -0.4494E-01
  0.6000E-03 -0.1800E+03  0.0000E+00  -0.5287E-01
.
.
.
  0.9550E-02 -0.1800E+03  0.0000E+00 -0.4173E+00
  0.9600E-02 -0.1800E+03  0.0000E+00 -0.4127E+00
  0.9650E-02 -0.1800E+03  0.0000E+00 -0.4082E+00
  0.9700E-02 -0.1800E+03  0.0000E+00 -0.4038E+00
  0.9750E-02 -0.1800E+03  0.0000E+00 -0.3997E+00
  0.9800E-02 -0.1800E+03  0.0000E+00 -0.3957E+00
  0.9850E-02 -0.1800E+03  0.0000E+00 -0.3921E+00
  0.9900E-02 -0.1800E+03  0.0000E+00 -0.3887E+00
  0.9950E-02 -0.1800E+03  0.0000E+00 -0.3853E+00
  0.1000E-01 -0.1800E+03  0.0000E+00 -0.3821E+00
```

Figure 11.20 Results from Program 11.7 example

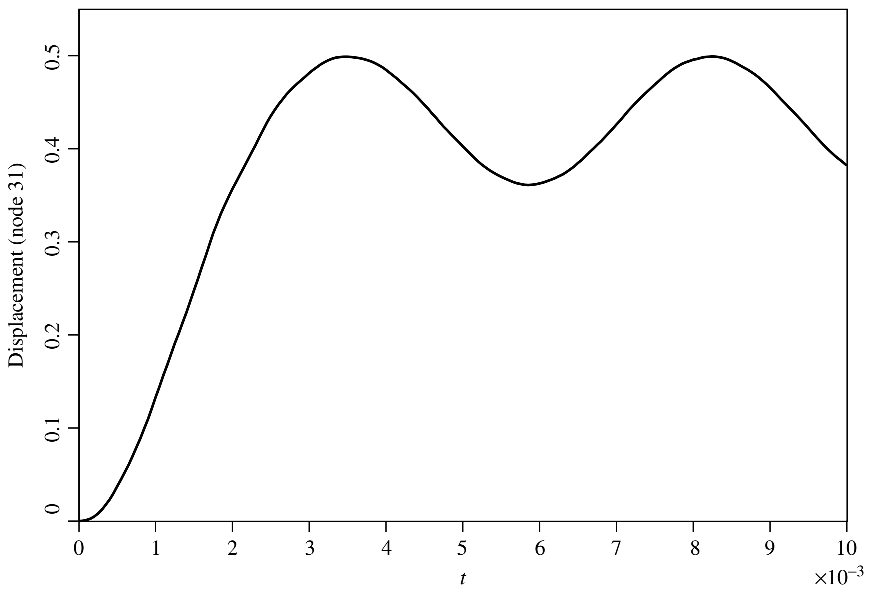


Figure 11.21 Displacement at node 31 versus time from Program 11.7 example

the centreline displacement of the beam as a function of time computed over the first 10,000 time steps. The development of permanent, plastic deformation is clearly demonstrated.

Glossary of variable names used in Chapter 11

Scalar integers:

`cg_iters`	pcg iteration counter
`cg_limit`	pcg iteration ceiling
`i`	simple counter
`iel`	simple counter
`idiag`	skyline bandwidth
`ifail`	warning flag from `bisect` subroutine
`iwp`	`SELECTED_REAL_KIND(15)`
`j`	simple counter
`jj`	simple counter
`k`	simple counter
`lnode`	loaded node number
`loaded_nodes`	number of loaded nodes
`lsense`	sense of freedom to be loaded at node `lnode`
`nband`	bandwidth of upper triangle
`ndof`	number of degrees of freedom per element
`ndim`	number of dimensions
`nels`	number of elements
`neq`	number of degrees of freedom in the mesh
`nip`	number of integrating points
`nlfp`	number of load function points
`nln`	number of loaded freedoms
`nmodes`	number of eigenvectors included in superposition
`nn`	number of nodes in the mesh
`nod`	number of nodes per elements
`nodof`	number of degrees of freedom per node
`nof`	number of output freedoms
`nprops`	number of material properties
`npri`	output printed every `npri` time steps
`np_types`	number of different property types
`nr`	number of restrained nodes
`nres`	node number at which time history is to be printed
`nst`	number of stress terms
`nstep`	number of calculation time steps
`nxe`	number of elements in x-direction
`nye`	number of elements in y-direction

Scalar reals:

`aa`	working variable
`alpha`	α from equations (3.22)
`area`	element area

`bb`	working variable
`beta`	Newmark time stepping parameter or β from equations (3.22)
`cg_tol`	pcg convergence tolerance
`c1`	working constant
`c2`	working constant
`c3`	working constant
`c4`	working constant
`det`	determinant of Jacobian matrix
`dr`	damping ratio
`dsbar`	invariant, $\overline{\sigma}$
`dtim`	calculation time step
`d4`	set to 4.0
`d6`	set to 6.0
`etol`	eigenvalue tolerance set to 1×10^{-30}
`f`	value of yield function or force vector
`fac`	measure of yield surface overshoot (f from equation (6.35))
`fk`	Rayleigh damping parameter on stiffness
`fm`	Rayleigh damping parameter on mass
`fmax`	maximum value of yield function f at each time step
`fnew`	value of yield function after stress increment
`f1`	temporary working variable
`f2`	temporary working variable
`gamma`	Newmark time stepping parameter
`k1`	working variable
`k2`	working variable
`lode_theta`	Lode angle, θ
`omega`	frequency of forcing term
`one`	set to 1.0
`penalty`	set to 1×10^{20}
`pt2`	set to 0.2
`pt25`	set to 0.25
`pt5`	set to 0.5
`sigm`	mean stress, σ_m
`theta`	time integration weighting parameter
`time`	holds elapsed time t
`two`	set to 2.0
`up`	holds dot product $\{\mathbf{R}\}_k^T \{\mathbf{R}\}_k$ from equations (3.22)
`zero`	set to 0.0

Scalar character:

`element`	element type

Scalar logicals:

`consistent`	set to `.TRUE.` if mass matrix is “consistent”
`cg_converged`	set to `.TRUE.` if pcg process has converged

Dynamic integer arrays:

`etype`	element property type vector
`g`	element steering vector
`g_g`	global element steering matrix
`g_num`	global element node numbers matrix
`kdiag`	diagonal term locations
`kdiag_l`	diagonal term locations on the left
`kdiag_r`	diagonal term locations on the right
`lf`	vector holding loaded freedoms
`lp`	vector holding output freedoms
`nf`	nodal freedom matrix
`node`	output/input nodes vector
`num`	element node number vector
`sense`	sense of output nodes

Dynamic real arrays:

`a`	accelerations
`acc`	accelerations at output freedoms
`al`	array holding all loading values at each calculation step
`a1`	temporary working vector
`bdylds`	self-equilibrating global body forces
`bee`	strain-displacement matrix
`bigk`	eigenvector matrix
`bload`	self-equilibrating element body forces
`b1`	temporary working vector
`coord`	element nodal coordinates
`cv`	damping matrix
`d`	displacements or vector used in equations (3.22)
`dee`	stress–strain matrix
`der`	shape function derivatives with respect to local coordinates
`deriv`	shape function derivatives with respect to global coordinates
`diag`	global lumped mass vector
`diag_precon`	diagonal preconditioner vector
`dis`	displacements at output freedoms
`d1x0`	“old” velocities
`d1x1`	“new” velocities
`d2x0`	“old” accelerations
`d2x1`	“new” accelerations
`ecm`	element consistent mass matrix
`eld`	element nodal displacements
`ell`	element lengths vector
`eload`	integrating point contribution to `bload`
`eps`	strain terms

`etensor`	holds running total of all integrating point strain terms
`fun`	shape functions
`f1`	left hand side matrix (stored as a skyline)
`g_coord`	nodal coordinates for all elements
`jac`	Jacobian matrix
`kd`	vector used to set up initial accelerations
`kh`	global stiffness vector
`km`	element stiffness matrix
`kp`	modified global "stiffness" matrix
`ku`	global stiffness matrix stored as upper triangle
`kv`	global stiffness matrix
`loads`	nodal loads and displacements
`mc`	global mass matrix used to set up initial accelerations
`mm`	element consistent mass matrix
`mv`	global consistent mass matrix
`p`	"descent" vector used in equations (3.22)
`pl`	plastic $[\mathbf{D}^p]$ matrix (6.29)
`points`	integrating point local coordinates
`prop`	element properties matrix
`rl`	input load function load values
`rrmass`	vector holding reciprocal of square root of lumped mass
`rt`	input load function time values
`sigma`	stress terms
`storkm`	holds element stiffness matrices
`stormm`	holds element mass matrices
`stress`	stress term increments
`tensor`	holds running total of all integrating point stress terms
`u`	vector used in equations (3.22)
`udiag`	transformed and untransformed eigenvectors
`v`	velocities
`val`	applied nodal load weightings
`vc`	vector used to set up initial accelerations
`vel`	velocities at output freedoms
`weights`	weighting coefficients
`x`	"old" solution vector
`xmod`	solutions to modal SDOF equations
`xnew`	"new" solution vector
`x0`	"old" displacements
`x1`	"new" displacements
`x_coords`	x-coordinates of mesh layout
`y_coords`	y-coordinates of mesh layout

Dynamic character array:

`mtype`	set to `'l'` for lumped mass and `'c'` for consistent

11.2 Exercises

1. The undamped beam shown in Figure 11.22 is initially at rest and subjected to a suddenly applied moment of one unit at its left support. Using a single finite element, a time step of 1 s and the Constant Acceleration Method ($\beta = 1/4$, $\gamma = 1/2$), estimate the rotation at both ends of the beam after 2 s.
(Ans: $\theta_1 = 0.284$, $\theta_2 = 0.0.036$)

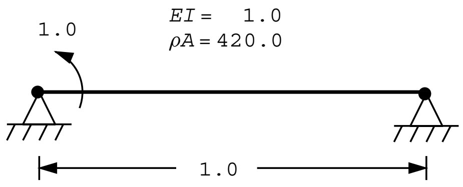


Figure 11.22

2. Repeat the previous question assuming 5% damping. Use Rayleigh damping by assuming the mass matrix damping parameter (f_m) equals zero. The fundamental natural frequency of the beam is $\omega_1 = 0.48$.
(Ans: If $f_m = 0$, then $\zeta_1 = \omega_1 f_k/2$, hence with $\zeta_1 = 0.05$ and $\omega_1 = 0.48$, $f_k = 0.208$. $\theta_1 = 0.254$, $\theta_2 = 0.0.015$)

3. The undamped propped cantilever shown in Figure 11.23 is initially at rest and subjected to a suddenly applied load at its mid-span. Using two finite elements, a time step of 1 s and the Linear Acceleration Method ($\beta = 1/6$, $\gamma = 1/2$), estimate the deflection under the load after 2 s.
(Ans: $u = 0.077$)

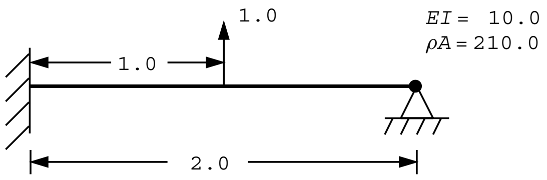


Figure 11.23

4. The undamped cantilever shown in Figure 11.24 is initially at rest and subjected to a suddenly applied load and moment at its tip. Using one finite element, a time step of 1 s and the Constant Acceleration Method ($\beta = 1/4$, $\gamma = 1/2$), estimate the deflection and rotation at the tip after 2 s.
(Ans: $u = 0.092$, $\theta = 0.653$)

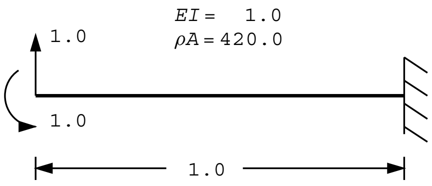


Figure 11.24

5. A simply supported beam of length, $L = 1$ and properties $EI = 1.0$, $\rho A = 3.7572$ is subjected to a constant transverse force, $P = 48$, which moves across the beam from the left to right support with a constant velocity, $U = 1$. By discretising the beam into 4 elements, compute the time dependent response of the centreline of the beam. Show that the centreline deflection reaches a maximum value that is approximately 1.743 times the deflection that would have been obtained if the load had been placed statically at the centre of the beam. Use equivalent fixed end moments and reactions to model the effect of the moving load at four locations within each element.

   Compare your result with the analytical solution at the centreline given by:

$$v = \frac{2PL^3}{\pi^4 EI}\left[\frac{\sin c\omega_1 t - c\sin\omega_1 t}{1-c^2}\right]$$

   where

$$\omega_1 = \frac{\pi^2}{L^2}\sqrt{\frac{EI}{\rho A}} \qquad \text{and} \qquad c = \frac{\pi U}{\omega_1 L}$$

References

Bathe KJ 1996 *Numerical Methods in Finite Element Analysis*, 3rd edn. Prentice Hall, Englewood Cliffs, N.J.

Key SW 1980 Transient response by time integration: A review of implicit and explicit operators. In *Advances in Structural Dynamics* (ed. Donea J). Applied Science, London, pp. 71–95.

Smith IM 1984 Adaptability of truly modular software. *Eng Comput* **1**(1), 25–35.

Warburton GB 1964 *The Dynamical Behaviour of Structures*. Pergamon Press, Oxford.

Wong SW, Smith IM and Gladwell I 1989 PCG methods in transient FE analysis Part II: Second order problems. *Int J Numer Methods Eng* **28**(7), 1567–1576.

12

Parallel Processing of Finite Element Analyses

12.1 Introduction

In the previous Chapters, serial finite element programs were listed for the solution of a wide variety of problems in engineering and science. As was mentioned in Chapter 1, analyses can be speeded up by vector processing as illustrated in Chapter 5, Program 5.6, but so far vector machines have not been widespread.

The more common approach, used in the majority of supercomputers at the moment, is parallel processing in which many standard (and therefore low cost) processors are linked together by fast communication networks. About 1,000 processors are typical at the time of writing.

However, supercomputers are still expensive, and an alternative at very low cost is to link together "clusters" of PCs by an Ethernet or similar low-cost communications network.

In this chapter, programs are listed which run in parallel on any system capable of supporting MPI (the "message passing interface" standard described in Chapter 1). This covers all current supercomputers (vector-parallel, shared memory, distributed memory) and PC clusters. Performance statistics are given for several such systems. OpenMP versions have also been successfully tested but are less portable.

The approach adopted is to take at least one program from each of the preceding Chapters 5 to 11 and parallelise it. The full range of algorithm types—linear static equilibrium, non-linear static equilibrium, eigenvalue and implicit and explicit transient, are covered making 10 programs in all.

The methodology assumes the same program running on every processor of a multiprocessor system, each processor usually operating on different data. From time to time a processor needs information that does not reside on that particular processor and has to be communicated to it via MPI (Pettipher and Smith, 1997).

Table 12.1 Effect of mesh subdivision in three dimensions

Mesh subdivision	Number of equations
$10 \times 10 \times 10$	12,580
$20 \times 20 \times 20$	98,360
$40 \times 40 \times 40$	777,520
$50 \times 50 \times 50$	1,514,900
$80 \times 80 \times 80$	6,182,240
$100 \times 100 \times 100$	12,059,800
$160 \times 160 \times 160$	49,305,280

The benefits sought are both in faster execution times (under perfect conditions n processors operating in parallel would decrease analysis time by a factor of n) and in ability to process larger problems, because the data can be distributed over the n processors. It is expected that most parallel processing by finite elements will involve problems that are spatially three-dimensional. Data demands for such analyses increase rather dramatically with problem size as shown in Table 12.1 which refers to an elastic cube meshed by 20-node elements (see Program 5.5). The cube has all four sides on rollers, a fixed base, and a free surface.

Thus, although a mesh of 160×160 elements in two dimensions might be considered modest, $160 \times 160 \times 160$ elements in three dimensions would need 400 Mb just to store a single vector of equations if the data were not distributed.

Using the program described first in this Chapter, a $160 \times 160 \times 160$ element elastic cube problem was solved in 90 min on a 64-processor system and in 10 min on a 512 processor system.

The serial programs closest to their parallel derivatives are given in Table 12.2 below:

Table 12.2 Serial and parallel program equivalence

Serial version	Parallel version	Remarks
Program 5.5	Program 12.1	20-node brick option, loaded freedoms only
Program 6.11	Program 12.2	loaded freedoms only
Program 7.5	Program 12.3	3D option
Program 8.3	Program 12.4	3D version
Program 8.4	Program 12.5	3D version
Program 9.2	Program 12.6	3D version
Program 9.5	Program 12.7	3D version
Program 10.4	Program 12.8	3D version
Program 11.5	Program 12.9	3D version
Program 11.6	Program 12.10	3D version

12.2 Differences between parallel and serial programs

As far as possible, the parallel programs copy their serial counterparts. For example comparison of Program 5.5 (serial) and Program 12.1 (parallel) will show that the element integration loop, beginning with label `gauss_pts_1` is exactly the same in both versions. Such a consistency indicates "local" or "embarrassingly parallelisable" sections of code.

When distributed arrays are involved, for example in the section following "pcg equation solution" the coding is identical with the exception of `_P` in the `DOT_PRODUCT` calls, the `_pp` appendage to array names (`r` becomes `r_pp` and so on) and the distributed convergence check `checon_par`.

In what follows, the differences between parallel and serial programs are described. These differences are common to all the parallelised programs.

12.2.1 Parallel libraries

Serial libraries `new_library` and `geometry_lib` perform the same tasks as `main` and `geom` in the earlier Chapters but are augmented by six others, as shown in Table 12.3.

Table 12.3 Parallel libraries

Library name	Usage
`precision`	Sets precision for `REAL` variables
`utility`	Sets various MPI routines for broadcasting, distributed `DOT_PRODUCT` etc
`mp_module`	Various MPI routines
`timing`	Routines to assist with performance evaluation
`global_variables1`	Designation of some widely used variables as "global", not declared elsewhere
`gather_scatter6`	MPI routines for collecting data from, or distributing it to, parallel processors

12.2.2 Global variables

In the serial programs, all variables were declared in all programs. In the parallel versions, some widely used variables are declared as "global." These are, with their meanings:

`ndof`	Number of degrees of freedom per element or sub-element
`nels`	Number of elements
`nels_pp`	Number of elements per processor (variable)
`neq`	Number of equations
`neq_pp`	Number of equations per processor (variable)
`ntot`	Total number of degrees of freedom per element
`ielpe`	Counter for element per processor

`iel_start`	Starting element number per processor
`ieq_start`	Starting equation number per processor
`numpe`	Local processor number or rank
`npes`	Number of "processing elements"
`ier`	MPI parameter for error checking

These variables must not be additionally declared.

12.2.3 MPI library routines

The same MPI routines are used in all programs. There are only a dozen or so that are listed below with their purposes (see Appendix F for further details of subroutines used in this Chapter and their arguments):

`MPI_INITIALIZE`	Initialise MPI (hidden)
`shutdown`	Close MPI: must appear
`DOT_PRODUCT_P`	Distributed version of dot product
`SUM_P`	Distributed version of array `SUM` N.B. We take the liberty of using capitals as if these were part of Fortran.
`norm_p`	Finds the l2 norm of a distributed vector
`find_pe_procs`	Finds how many processors are being used
`calc_nels_pp`	Finds number of elements per processor (variable)
`calc_neq_pp`	Finds number of equations per processor (variable)
`reduce`	Finds maximum of a distributed integer variable
`make_ggl`	Builds distributed `g` vectors (see Section 3.7.10 for description of `g`)
`gather`	See Section 12.2.8
`scatter`	See Section 12.2.8
`checon_par`	Convergence check for distributed vectors
`reindex_fixed_nodes`	See Section 12.2.9
`bcast_inputdata_pxxx`	See Section 12.2.5.

12.2.4 The _pp appendage

Distributed arrays and their upper bounds carry the appendage `_pp`. A difference from the serial programs is that it is more convenient to begin array addresses at 1 rather than 0. So the serial `p(0:neq)` becomes `p_pp(neq_pp)` in parallel.

12.2.5 Reading and writing

The simple approach adopted here is that data are read, and results written, on a single (not necessarily the same) processor. Data, having been read on one processor, are then broadcast to all other processors by MPI routines such as `bcast_inputdata_p121` that are unique to each parallel program as the `_p121` appendage implies. For results to be written easily, it is necessary to find which processor contains the desired quantities. In the programs in this Chapter a single output processor is identified and used.

12.2.6 Problem-specific boundary condition routines

In order to assess the benefits of parallelism it is necessary to be able to refine meshes readily for the same basic analysis. For this reason simple cuboidal geometries are used exclusively in this Chapter, and for these it is possible to introduce boundary conditions via problem-specific routines.

For example Programs 12.1 and 12.2 respectively analyse a cuboid of elastic or elasto-plastic material made up from 20-node bricks as shown in Figure 12.1.

Note that the z axis is the vertical rather than the y axis as in Program 5.5. All four vertical faces are on rollers; the front and left are planes of symmetry and the back and right are external boundaries. The base is completely fixed and the top completely free (except at the roller edges).

To simplify the loading, it is assumed to occupy a square patch extending to one-fifth of the cube surface in the x- and y-directions. It is therefore assumed that numbers of elements in the x- and y-directions are multiples of 5. Subroutine `cube_bc20` applies the appropriate boundary conditions and subroutine `loading` the appropriate loading.

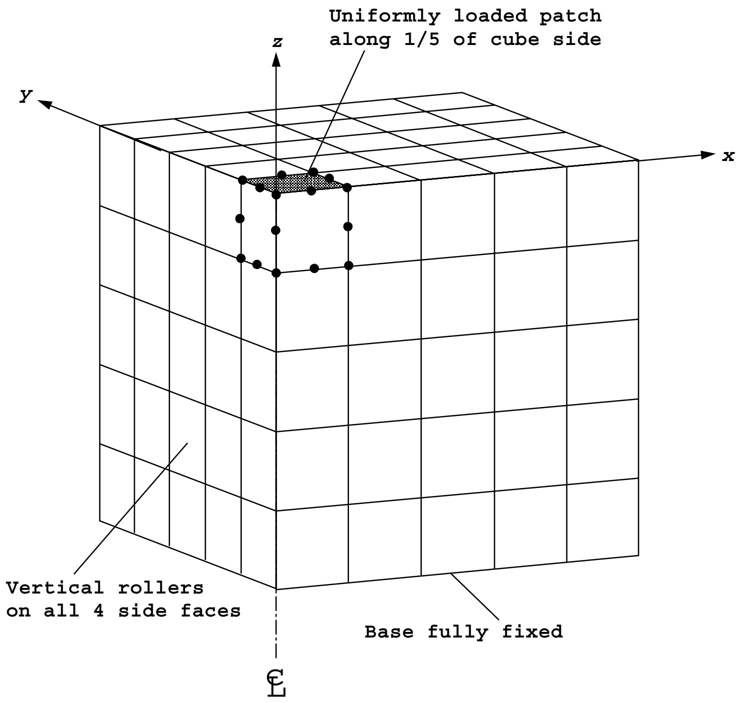

Uniformly loaded patch along 1/5 of cube side
z
y
x
Vertical rollers on all 4 side faces
Base fully fixed
℄

Figure 12.1 Mesh for Programs 12.1 and 12.2

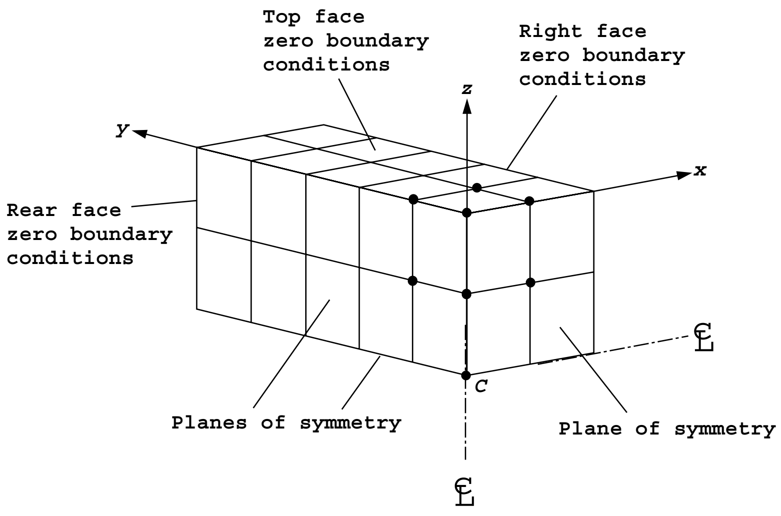


Figure 12.2 Mesh for Programs 12.3, 12.4, 12.5

Programs 12.3, 12.4, and 12.5 analyse a cuboidal box of 8-node elements as shown in Figure 12.2.

The boundary conditions assumed are that the top, back and right hand faces of the box have dependent variable of zero while the left, front, and base are planes of symmetry. In Program 12.3 potential or flux (see Program 7.4) can be specified at C, the centre of the box, and the results are printed for the centre and a few nodes to the right. In Programs 12.4 and 12.5, which involve transient analyses, the variation of dependent variable with time is printed at C only, for a given (uniform) initial distribution. Subroutine `box_bc8` applies the appropriate boundary conditions.

Program 12.6 conducts a Navier–Stokes analysis for the classic lid-driven cavity which is assumed to be cuboidal (no symmetry in this case). Thus the velocities on all faces are fixed in x, y and z, except for the top face that is driven with constant velocity in the x-direction but otherwise fixed. To avoid a singularity in the pressure field, pressure is assumed to be zero along the left hand edge of the top of the box (analogous to Programs 9.1 and 9.2 where the top left hand corner had zero pressure). Subroutine `ns_cube_bc20` applies the appropriate boundary conditions. A typical mesh is shown in Figure 12.3.

Program 12.7 analyses coupled consolidation of a cuboid of porous elastic material. The boundary conditions on the solid part are the same as those in Figure 12.1 while the fluid pressure is zero on the top surface only. Subroutine `biot_loading` again assumes a square patch extending to one-fifth of the surface edge length. Subroutine `biot_cube_bc20` supplies the appropriate boundary conditions. A typical mesh is shown in Figure 12.4.

The remaining programs in the Chapter, Programs 12.8, 12.9, 12.10 all analyse cuboidal cantilevers of elastic or elasto-plastic material as shown in Figures 12.5 and 12.6.

The elements can be 8-node (Program 12.8) or 20-node (Programs 12.9, 12.10) bricks and the front x-z face is completely fixed in x, y, and z. In this simple case the number of restrained nodes, `nr`, is easily calculated and the appropriate boundary condition applied.

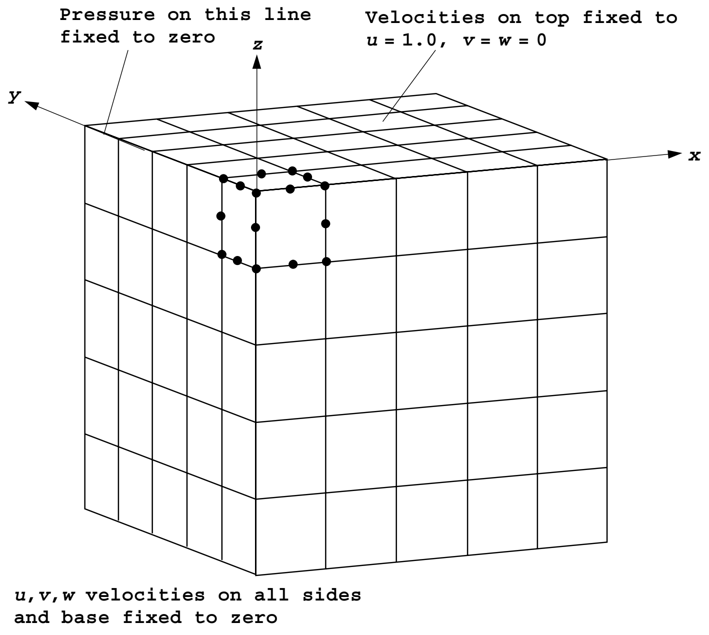

Pressure on this line fixed to zero
z
Velocities on top fixed to $u = 1.0$, $v = w = 0$
y
x
u, v, w velocities on all sides and base fixed to zero

Figure 12.3 Mesh for Program 12.6

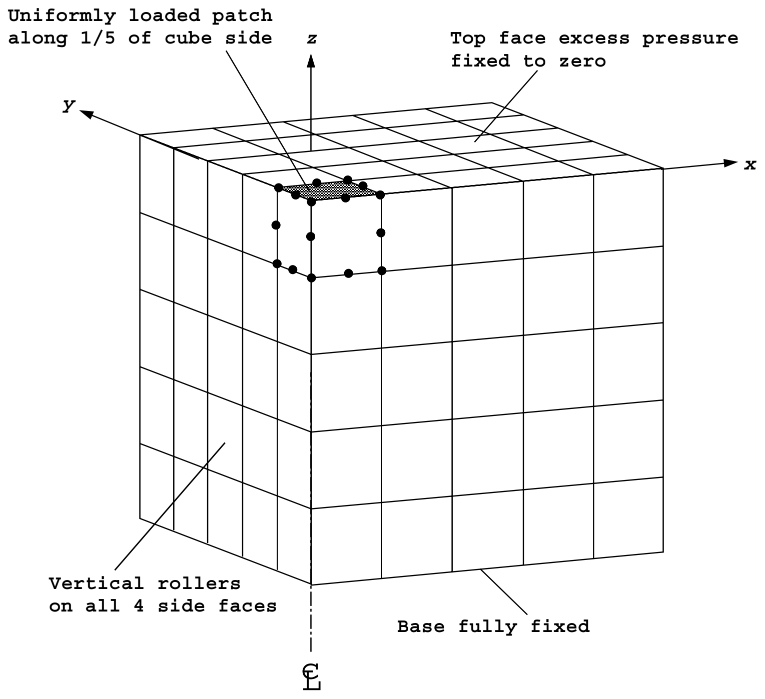

Uniformly loaded patch along 1/5 of cube side
z
Top face excess pressure fixed to zero
y
x
Vertical rollers on all 4 side faces
Base fully fixed
℄

Figure 12.4 Mesh for Program 12.7

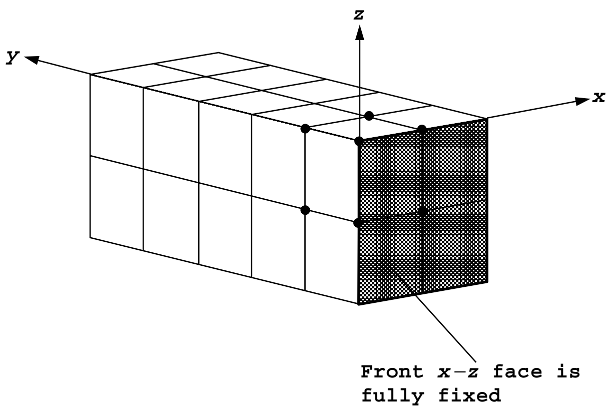


Figure 12.5 Mesh for Program 12.8

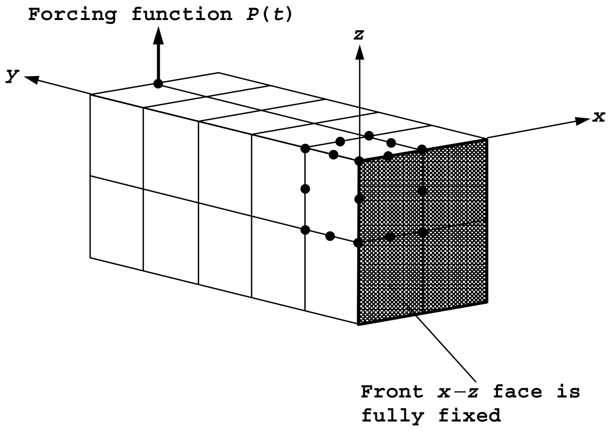


Figure 12.6 Mesh for Programs 12.9 and 12.10

For Programs 12.9 and 12.10, in which a load is applied, an even number of elements in the x-direction is assumed, so that the load can be applied at the mid-point of the top of the free-end face as shown in Figure 12.6.

12.2.7 `rest` instead of `nf`

In the serial programs a “node freedom array,” `nf` (see Section 3.7.10) was employed. This array contained information about every node in the mesh, whether restrained or not. In very large problems this is wasteful because the restrained (usually boundary) nodes become a smaller and smaller proportion of the total. For this reason the parallel programs use an array `rest` instead of `nf`. Exactly the same restraint information is created as would be the case for `nf` but instead of being read in, it is created by the boundary condition routines described in the previous section. Thus routines like `cube_bc20` return `rest` from a knowledge of the number of elements in each direction and the problem-specific boundary conditions. Following its creation, `rest` is often rearranged using subroutines `rearrange` or `rearrange_2` before being used in routines to calculate the steering vector `g` from the restraint data. Because there is now no `nf`, replacements for subroutine `num_to_g` which was used in all serial programs are necessary. The replacement routine is `find_g3` in Programs 12.1, 12.2, 12.6, 12.7, `find_g4` in Programs 12.3, 12.4, 12.5, and `find_g` in Programs 12.8, 12.9, 12.10. The different versions are necessary because rather large volumes of data are being searched.

It should be re-emphasised that the number of restrained nodes, `nr`, is now explicitly calculated in each parallel program and does not have to be input.

12.2.8 Gathering and scattering

This is done very neatly in the serial programs using the power of Fortran 95 as described in Section 1.9.4. Thus a typical gather-matrix multiply-scatter loop around the elements, the core of EBE iteration methods, might read:

```
elements_2: DO iel=1,nels
  g=g_g(:,iel)
  pmul=p(g)
  utemp=MATMUL(km,pmul)
  u(g)=u(g)+utemp
END DO elements_2
```

In parallel, `u` and `p` are distributed as `u_pp` and `p_pp` while `utemp` and `pmul` are distributed as two-dimensional arrays `utemp_pp` and `pmul_pp` that hold all the components of `utemp` and `pmul` for that processor.

Thus the parallel loop becomes:

```
CALL gather(p_pp,pmul_pp)
elements_2: DO iel=1,nels_pp
  utemp_pp(:,iel)=MATMUL(km,pmul_pp(:,iel))
END DO elements_2
CALL scatter(u_pp,utemp_pp)
```

12.2.9 Reindexing

When loads are read in or displacements fixed, their global freedom numbers are specified. In parallel, the appropriate equations are distributed across the processors in some way and so the appropriate indexing must be found using `reindex_fixed_nodes`.

12.2.10 Domain composition

For parallel FE processing, pieces of a large mesh have to be allocated to the processors. These pieces are traditionally called *subdomains*. It is also traditional to speak of the whole mesh or "domain" being "decomposed" into its constituent subdomains. This use of words implies that a large mesh (domain) is assembled, in principle by the global assembly techniques described in the previous Chapters, and that the resulting global equations are "decomposed" or torn apart into smaller pieces. Indeed some early implementations of this process were called *diakoptics*, implying a cutting up procedure. "Substructuring," and "block" and "frontal" methods are similar variants, the main application being the solution, by elimination techniques, of the very large sets of linear algebraic equations which govern the static equilibrium of linear and non-linear FE systems.

All of this suggests the parallelisation of the "global" strategy used in earlier Chapters of this book. When Gaussian elimination methods are used, elimination can proceed independently on equations relating to a particular subdomain, stored on a particular processor, until the boundaries of the subdomain are reached. Then communication is necessary between processors. This can be a rather complicated procedure, and a large literature has developed around the theme of optimising subdomain distributions. Often equations are solved directly within subdomains but iteratively at the boundaries.

In contrast, in this book a simple approach is used, based on the element-by-element methods used in previous Chapters. In these, no global equation system matrices are ever constructed and therefore it is more meaningful to speak of "domain composition" rather than "decomposition." But of course subdomains have to be identified.

In the element-by-element technique, see Section 3.5, the essential operations that involve inter-processor communication are the "gather, matrix multiply, scatter" procedure typified by equations (3.23) and the dot products typified by equations (3.22), (3.28) and (3.29). Clearly different subdomain distributions will affect the amount of communcation

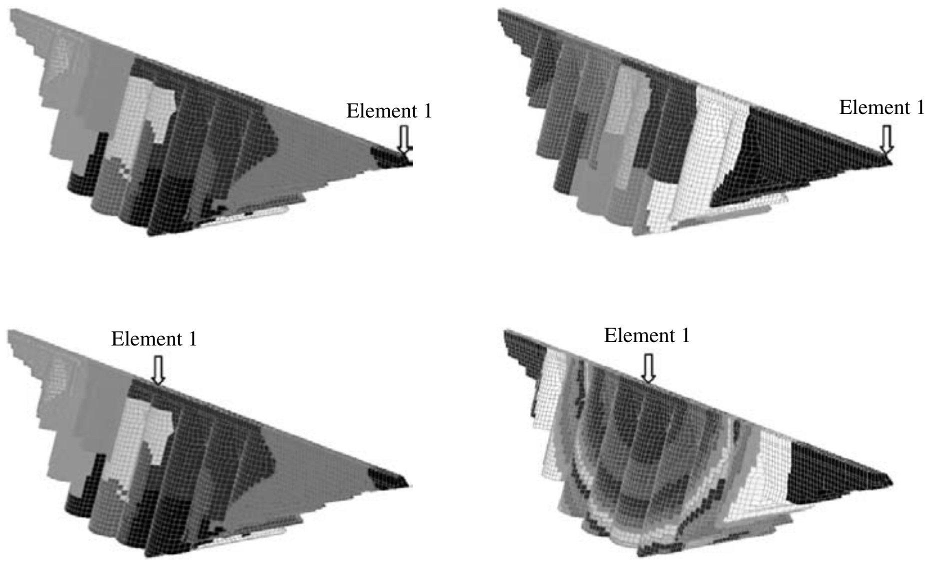


Figure 12.7 Alternative domain compositions

involved. Margetts (2002) (see also Smith and Margetts, 2003) has shown that there is certainly no simple solution to this problem for complicated domains. Four different domain compositions, shown in Figure 12.7 for the case of a buttress dam, led to broadly similar analysis times on a supercomputer. Some compositions lead to fewer large messages being exchanged between processors and others to more short messages.

For the cuboidal meshes used in this Chapter a "naive" composition by slices on *x*-*z* planes is used.

12.2.11 Load balancing

In our "naive" domain compositions the "load" on processors (the amount of computation they do) is almost perfectly distributed or "balanced" by assigning almost equal numbers of elements, `nels_pp` and equations, `neq_pp` to each processor. Note that if the mesh is too small for the number of processors requested, the analysis will not continue.

We are now in a position to describe the parallel programs in detail.

Program 12.1 Three dimensional analysis of an elastic solid. Compare Program 5.5.

```
PROGRAM p121
!-----------------------------------------------------------------------
!      Program 5.5 three dimensional analysis of an elastic solid
!      using 20-node brick elements, preconditioned conjugate gradient
!      solver; only integrate one element, diagonal preconditioner
!      diag_precon; parallel version ; loaded  freedoms only ; cube_bc
!-----------------------------------------------------------------------
  USE new_library; USE  geometry_lib; USE precision; USE utility
  USE mp_module  ; USE  timing    ; USE global_variables1
  USE gather_scatter6; IMPLICIT NONE
 !  ndof, nels, neq , ntot  are now global variables - not declared
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=3,nod=20,nst=6,loaded_freedoms,     &
   i,j,k,ndim=3,iters,limit,iel,num_no,no_index_start,neq_temp,nn_temp,nle
 REAL(iwp)::aa,bb,cc,e,v,det,tol,up,alpha,beta,q
 CHARACTER(LEN=15)::element= 'hexahedron';    LOGICAL :: converged
!-------------------------- dynamic arrays-------------------------------
 REAL(iwp),ALLOCATABLE :: points(:,:),dee(:,:),coord(:,:), weights(:),    &
     val(:),p_g_co_pp(:,:,:), jac(:,:), der(:,:), deriv(:,:),bee(:,:),    &
     km(:,:),eld(:),eps(:),sigma(:),eld_pp(:,:),diag_precon_pp(:),p_pp(:),&
     r_pp(:),x_pp(:), xnew_pp(:),u_pp(:),pmul_pp(:,:),utemp_pp(:,:),      &
     d_pp(:),diag_precon_tmp(:,:)
 INTEGER, ALLOCATABLE :: rest(:,:), g(:), num(:), g_num_pp(:,:)   ,       &
     g_g_pp(:,:),no(:),no_local_temp(:),no_local(:)
!-------------------------input and initialisation-------------------------
 timest(1) = elap_time( )   ;    CALL find_pe_procs(numpe,npes)
 IF (numpe==npes) THEN
  OPEN (10,FILE='p121.dat',STATUS=    'OLD',ACTION='READ')
  READ (10,*) nels,nxe,nze,nip,aa,bb,cc,e,v, tol,limit
 END IF
 CALL bcast_inputdata_p121(numpe,npes,nels,nxe,nze,nip,aa,bb,cc,          &
                    e,v,tol,limit)
 CALL calc_nels_pp ; ndof=nod*nodof ; ntot=ndof
 nn_temp = 0; neq_temp = 0  ; nye = nels/nxe/nze  ; nle = nxe/5
 nr = ((2*nxe+1)*(nze+1)+(nxe+1)*nze)*2+((2*nye-1)*nze+(nye-1)*nze)*2     &
      + (2*nye-1)*(nxe+1) + (nye-1)*nxe
 loaded_freedoms = 3*nle*nle + 4*nle + 1
 ALLOCATE (points(nip,ndim),dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),  &
          der(ndim,nod),deriv(ndim,nod),eld_pp(ntot,nels_pp),             &
          bee(nst,ntot),km(ntot,ntot),eld(ntot),eps(nst),sigma(nst),      &
          g(ntot),pmul_pp(ntot,nels_pp),utemp_pp(ntot,nels_pp),num(nod),  &
          p_g_co_pp(nod,ndim,nels_pp),g_num_pp(nod,nels_pp),weights(nip),&
          g_g_pp(ntot,nels_pp),no(loaded_freedoms),rest(nr,nodof+1),      &
          val(loaded_freedoms),no_local_temp(loaded_freedoms),            &
          diag_precon_tmp(ntot,nels_pp))
 CALL cube_bc20(nxe,nye,nze,rest)  ; CALL rearrange(rest)
 CALL loading( nxe,nze,nle,no,val); val = - val * aa * bb / 12._iwp
 CALL deemat(dee,e,v);CALL sample(element,points,weights); ielpe=iel_start
!---------- loop the elements for global cordinates etc ------------------
     elements_0: DO iel = 1 , nels_pp
            CALL geometry_20bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
            CALL find_g3(num,g,rest) ; g_num_pp(:,iel) = num
            p_g_co_pp(:,:,iel) = coord; g_g_pp(:,iel) = g ; ielpe=ielpe+1
            i = MAXVAL(g); j = MAXVAL(num)
            IF(i>neq_temp)neq_temp = i  ;  IF(j>nn_temp)nn_temp = j
     END DO elements_0
   neq = reduce(neq_temp);   nn = reduce(nn_temp)
   CALL calc_neq_pp ;   CALL make_ggl(g_g_pp)
   ALLOCATE(p_pp(neq_pp),r_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp),       &
           u_pp(neq_pp),diag_precon_pp(neq_pp),d_pp(neq_pp))
           r_pp = .0_iwp; p_pp = .0_iwp; x_pp = .0_iwp; xnew_pp = .0_iwp
           diag_precon_pp=.0_iwp; diag_precon_tmp=.0_iwp
!------ element stiffness integration and build the preconditioner-------
      iel=1;CALL geometry_20bxz(iel,nxe,nze,aa,bb,cc,coord,num);km=0.0_iwp
      gauss_pts_1:  DO i=1,nip
               CALL shape_der (der,points,i) ; jac = MATMUL(der,coord)
               det = determinant(jac); CALL invert(jac)
               deriv = MATMUL(jac,der) ; CALL beemat (bee,deriv)
               km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
      END DO gauss_pts_1
     elements_1: DO iel = 1,nels_pp
              DO k=1,ndof;diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)  &
                                                 +km(k,k); END DO
     END DO elements_1
   CALL scatter(diag_precon_pp,diag_precon_tmp);DEALLOCATE(diag_precon_tmp)
     IF(numpe==1)THEN
      OPEN (11,FILE='p121.res',STATUS='REPLACE',ACTION='WRITE')
      WRITE(11,'(A,I5,A)') "This job ran on ", npes ," processors"
      WRITE(11,'(A)') "Global coordinates and node numbers "
      DO i = 1 , nels_pp,nels_pp-1
             WRITE(11,'(A,I8)')"Element ",i ; num = g_num_pp(:,i)
        DO k = 1,nod;WRITE(11,'(A,I7,3E12.4)')                           &
             "  Node ",num(k),p_g_co_pp(k,:,i); END DO
      END DO
      WRITE(11,'(A,3(I8,A))')"There are ",nn," nodes",nr,                &
                            " restrained and ", neq," equations"
      WRITE(11,*) "Time after setup is   :", elap_time( ) - timest(1)
     END IF
```

```
!-------------------- get starting r------------------------------------
    IF(loaded_freedoms>0) THEN
     CALL reindex_fixed_nodes                                          &
                 (ieq_start,no,no_local_temp,num_no,no_index_start)
     ALLOCATE(no_local(1:num_no)) ; no_local = no_local_temp(1:num_no)
     DEALLOCATE(no_local_temp)
        DO i = 1 , num_no
         r_pp(no_local(i)-ieq_start+1) = val(no_index_start + i - 1)
        END DO
    END IF                  ; q = SUM_P(r_pp)
    IF(numpe==1)  WRITE(11,'(A,E12.4)') "The total load is ", q
    diag_precon_pp=1._iwp/diag_precon_pp;d_pp=diag_precon_pp*r_pp;p_pp=d_pp
!--------------------preconditioned c. g. iterations----------------------
       iters = 0
     iterations  :      DO
       iters=iters+1; u_pp=0._iwp;pmul_pp=.0_iwp;CALL gather(p_pp,pmul_pp)
       elements_2 : DO iel = 1, nels_pp
                       utemp_pp(:,iel) = MATMUL(km,pmul_pp(:,iel))
!                      CALL dgemv('n',ntot,ntot,1.0,km,ntot,            &
!                                 pmul_pp(:,iel),1,0.0,utemp_pp(:,iel),1)
       END DO elements_2  ;       CALL scatter(u_pp,utemp_pp)
!--------------------------pcg equation solution--------------------------
           up=DOT_PRODUCT_P(r_pp,d_pp);alpha= up/ DOT_PRODUCT_P(p_pp,u_pp)
           xnew_pp = x_pp + p_pp* alpha ; r_pp=r_pp - u_pp*alpha
           d_pp = diag_precon_pp*r_pp; beta=DOT_PRODUCT_P(r_pp,d_pp)/up
           p_pp=d_pp+p_pp*beta
           CALL checon_par(xnew_pp,x_pp,tol,converged,neq_pp)
           IF(converged .OR. iters==limit) EXIT
     END DO iterations
 IF(numpe==1)THEN
   WRITE(11,'(A,I5)')"The number of iterations to convergence was  ",iters
   WRITE(11,'(A,E12.4)')"The central nodal displacement is  :",xnew_pp(1)
 END IF
!-------------------recover stresses at centroidal gauss-point------------
  nip=1;  points = .0_iwp  ; eld_pp = .0_iwp; CALL gather(xnew_pp,eld_pp)
  iel = 1;    coord= p_g_co_pp(:,:,iel);  eld=eld_pp(:,iel)
 IF(numpe==1)WRITE(11,'(A)')"The Centroid point stresses for element 1 are"
     gauss_pts_2: DO i= 1 , nip
       CALL shape_der(der,points,i); jac= MATMUL(der,coord)
       CALL invert (jac);   deriv= MATMUL(jac,der) ; CALL beemat(bee,deriv)
       eps   = MATMUL (bee,eld)  ; sigma = MATMUL (dee,eps)
       IF(numpe==1.AND.i==1)THEN
        WRITE(11,'(A,I5)') "Point  ",i   ; WRITE(11,'(6E12.4)')  sigma
       END IF
     END DO gauss_pts_2
 IF (numpe==1) WRITE(11,*)"This analysis took  :", elap_time( )-timest(1)
  CALL shutdown( )
 END PROGRAM p121
```

Scalar integers:

`i`	simple counter
`iel`	element counter
`iters`	iteration counter
`j`	simple counter
`k`	simple counter
`limit`	iteration ceiling

`loaded_freedoms`	number of loaded freedoms
`ndim`	number of dimensions
`neq_temp`	temporary equation sum
`nip`	number of integrating points
`nle`	number of loaded elements (side of square)
`nn`	number of nodes in the mesh
`nn_temp`	temporary node sum
`nod`	number of nodes per element
`nodof`	number of degrees of freedom per node
`no_index_start`	start address of loaded/fixed freedoms
`nr`	number of restrained nodes
`nst`	number of stress/strain terms
`num_no`	number of processors holding load/displacement data
`nxe`	number of elements in x direction
`nye`	number of elements in y direction
`nze`	number of elements in z direction

Scalar reals:

`aa`	x dimension of elements
`alpha`	pcg parameter
`beta`	pcg parameter
`bb`	y dimension of elements
`cc`	z dimension of elements
`det`	determinant of Jacobian matrix
`e`	Young's Modulus
`q`	total load
`tol`	convergence tolerance
`up`	pcg parameter
`v`	Poisson's Ratio

Scalar characters:

`element`	element type

Scalar logicals:

`converged`	set to `.TRUE.` if solution converged

Dynamic integer arrays:

`g`	element steering vector
`g_g_pp`	distributed global steering matrix
`g_num_pp`	distributed global element node numbers matrix
`no`	freedoms to be loaded/fixed
`no_local`	local (processor) freedoms
`no_local_temp`	temporary store
`num`	element node numbers
`rest`	node freedom restraints

Dynamic real arrays:

`bee`	strain-displacement matrix

`coord`	element nodal coordinates
`dee`	stress–strain matrix
`der`	derivatives wrt local coordinates
`deriv`	derivatives wrt global coordinates
`diag_precon_pp`	distributed diagonal preconditioning matrix
`diag_precon_tmp`	temporary store
`d_pp`	distributed pcg vector
`eld`	element nodal displacements
`eld_pp`	distributed nodal displacements
`eps`	element strains
`jac`	Jacobian matrix
`km`	element stiffness matrix
`points`	integrating point local coordinates
`p_g_co_pp`	distributed nodal coordinates
`p_mul_pp`	gather-scatter matrix
`p_pp`	distributed pcg vector
`r_pp`	distributed pcg vector
`sigma`	element stresses
`store_pp`	temporary storage
`utemp_pp`	gather-scatter matrix
`u_pp`	distributed pcg vector
`val`	prescribed load/displacement values
`weights`	weighting coefficients
`xnew_pp`	distributed pcg vector
`x_pp`	distributed pcg vector

After declarations, timing is started and the identification number of each processor, `numpe`, and the total number of processors used, `npes` is calculated using `find_pe_procs`. Data are read on the last processor only (note that compared with the serial version the input does not need restraint and loading information). These data have then to be broadcast to all other processors using `bcast_inputdata_p121`.

Then the number of elements per processor can be calculated by `calc_nels_pp`. The number of elements in the y-direction for simple cuboids, `nye`, is calculated, followed by the number of loaded elements, `nle`, assumed to be `nxe/5` in this case. The total number of restrained nodes, `nr`, can be calculated and the number of loaded freedoms, `loaded_freedoms`.

After array allocation, the restraint array `rest` can be calculated using `cube_bc20` and rearranged. Subroutine `loading` returns the numbers of the loaded freedoms, `no`, and their weighted values, `val`. These are then adjusted for element size (`aa, bb`).

Loop `elements_0` is the parallel equivalent of serial loop `elements_1`, `iel_start` being the number of the first element stored on each processor. Integers `neq_temp` and `nn_temp` calculate the largest equation and node number respectively found on each processor. After the loop, which stores node numbers, nodal coordinates and freedom numbers for every element, `reduce` produces the global numbers of equations `neq` and nodes `nn`. There are about three quarters of a million equations in this analysis.

The numbers of equations to be distributed to each processor can then be calculated using `calc_neq_pp` and the freedom information using `make_ggl`. Distributed equation arrays can then be allocated.

The section commented "element stiffness integration etc" down to `END DO gauss_pts_1` is identical in parallel and serial versions, but building the diagonal preconditioner in parallel involves a scatter operation.

Information about the analysis is written on processor 1. The next section of coding has to relocate the global loading `val` entries to the appropriate processors, using `reindex_fixed_nodes`, print out the total load and invert the preconditioner.

The section commented "preconditioned cg iterations" is the parallel equivalent of the similarly annotated section in Program 5.5, involving gather and scatter as described in Section 12.2.8. In a similar way the section commented "pcg equation solution" mirrors the serial version in an obvious way. Only the centreline vertical displacement of the elastic cuboid is printed.

Finally the stress recovery section, involving loop labelled `gauss_pts_2`, uses exactly the same coding as the serial version but the stresses are only printed for the central surface element at the first Gauss point.

The example analysed is an elastic cube with a uniform pressure of unity on a square patch at the centre of the cube. Data are listed as Figure 12.8 and results as Figure 12.9. The vertical deflection is seen to be −0.03428 units and the vertical stress under the load −0.999 (compared to −1.000 applied).

In all, the parallel program is about 50% longer than its serial counterpart. Two salient aspects of performance are shown in Figures 12.10 and 12.12. The success of iterative methods clearly depends on the number of iterations for convergence as a proportion of problem size (Smith and Wang, 1998). Figure 12.10 shows that the iteration

```
nels   nxe   nze   nip
64000  40    40     8

aa     bb    cc    e      v
0.25   0.25  0.25  100.0  0.3

tol       limit
1.0e-5    2000
```

Figure 12.8 Data for Program 12.1 example

```
This job ran on    32 processors

There are   270641 nodes   24161 restrained and   777520 equations
 Time after setup is   : 0.600000000000363798
The total load is  -0.4000E+01
The number of iterations to convergence was    568
The central nodal displacement is  : -0.3428E-01
The Centroid point stresses for element 1 are
Point     1
 -0.7032E+00 -0.7032E+00 -0.9994E+00  0.3700E-03  0.3846E-03  0.3846E-03
 This analysis took  : 83.4400000000005093
```

Figure 12.9 Results from Program 12.1 example

Problem Size	Iterations to convergence	Iters/Size
12,000	156	1.30e-2
98,000	297	3.03e-3
777,000	568	7.31e-4
1,514,000	704	4.65e-4
2,613,000	838	3.21e-4
6,812,000	1049	1.54e-4
12,059,000	1297	1.07e-4

Figure 12.10 Iterations to convergence against problem size: Program 12.1

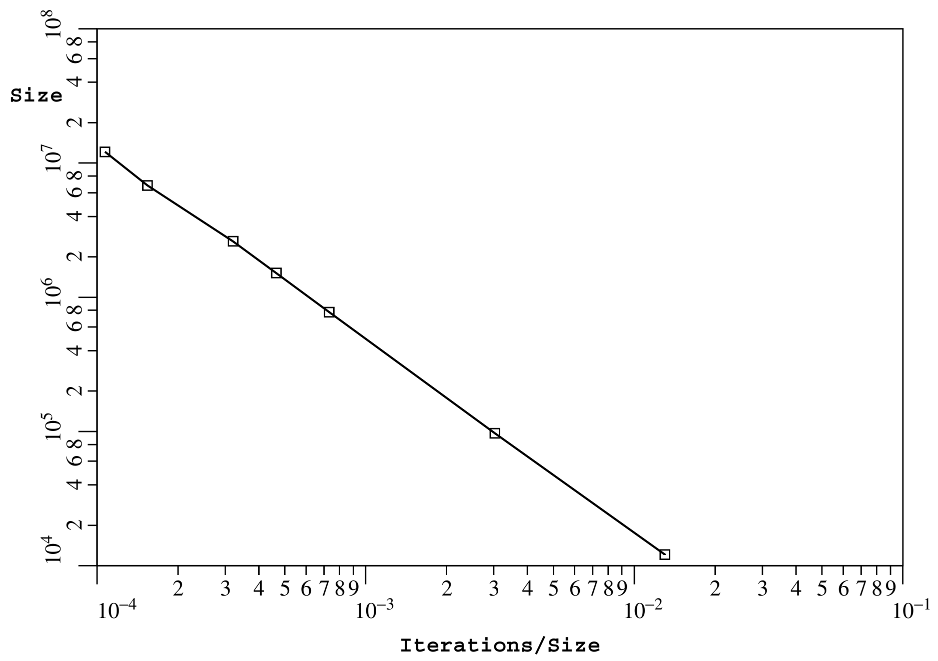


Figure 12.11 Iterations/size versus Problem size: Program 12.1

count tends to decrease sharply with problem size, and this is illustrated graphically in Figure 12.11.

Figure 12.12 shows, for a given problem size, the analysis time on a “supercomputer” decreasing but levelling off when a sufficient number of processors has been reached. Thus for any given problem size there comes a point where adding extra processors brings no benefit. This limiting number of processors increases with problem size. Conversely, a problem can be so small that parallelisation does not bring any benefit at all in terms of analysis time. However, in some cases, distribution of data may allow parallel processing of a job that could not be run at all serially due to memory limitations. Figure 12.13 shows speed-up vs. number of processors for larger data sets (SGI Origin 3000 computer involving up to 12 million equations).

Mesh	No of Processors	Analysis Time(secs)[Using BLAS]
20x20x20	1	148
	4	38.7
	8	20.0
	16	11.1
	32	6.9
40x40x40	16	163
	32	84
50x50x50	32	208[70]
	64	121
60x60x60	32	439
80x80x80	32	1408
	64	703[289]
100x100x100	64	[787]

Figure 12.12 Performance statistics: Program 12.1 (IBM SP2)

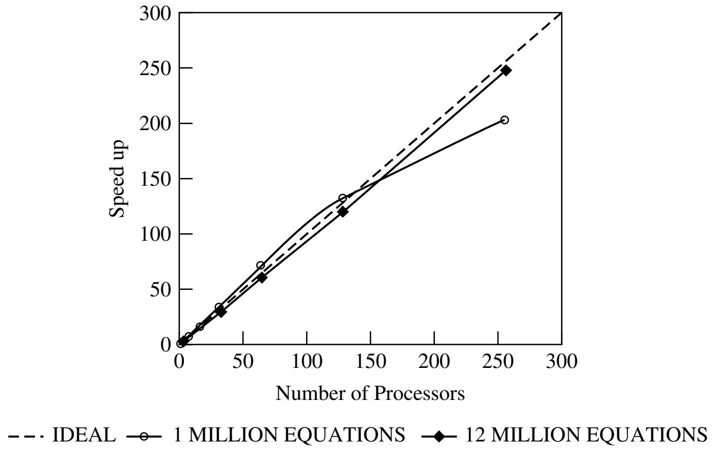


Figure 12.13 Speedup versus Number of processors: Program 12.1 (SGI Origin 3000)

Program 12.2 Three dimensional analysis of an elasto-plastic (Mohr–Coulomb) solid. Compare Program 6.11.

```
    PROGRAM p122
!-------------------------------------------------------------------
!    Program 6.11 three-d strain of an elastic-plastic(Mohr-Coulomb) solid
!    using 20-node brick elements; viscoplastic strain method,pcg parallel
!    cube_bc  ; pick up current x not x = .0  ; load control
!--------------------------------------------------------------------
 USE new_library; USE geometry_lib; USE precision; USE utility
 USE mp_module; USE timing;      USE global_variables1
 USE gather_scatter6; IMPLICIT NONE
! ndof,nels,neq,ntot are global - must not be declared
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=3,nod=20,nst=6,i,j,k,iel,plasiters,&
          plasits, cjiters,cjits,cjtot, incs,iy,ndim=3,loaded_freedoms,  &
```

```
         num_no,no_index_start,neq_temp,nn_temp , nle
 LOGICAL :: plastic_converged,cj_converged
 CHARACTER (LEN=15) :: element='hexahedron'
 REAL(iwp)::e,v,det,phi,c,psi,dt,f,dsbar,dq1,dq2,dq3,lode_theta,presc,   &
          sigm,pi,snph,cons,aa,bb,cc,plastol,cjtol,up,alpha,beta,big
!-------------------------- dynamic arrays----------------------------
 REAL(iwp),ALLOCATABLE ::loads_pp(:),points(:,:),bdylds_pp(:),           &
     evpt_pp(:,:,:),pmul_pp(:,:),dee(:,:),coord(:,:),jac(:,:),weights(:),&
     oldis_pp(:),der(:,:),deriv(:,:),bee(:,:),km(:,:),eld(:),eps(:),     &
     sigma(:),bload(:),eload(:),erate(:),p_g_co_pp(:,:,:),evp(:),devp(:),&
     m1(:,:),m2(:,:),m3(:,:),flow(:,:),storkm_pp(:,:,:),r_pp(:),val(:),  &
     tensor_pp(:,:,:),stress(:),totd_pp(:),qinc(:),p_pp(:),x_pp(:),      &
     xnew_pp(:),u_pp(:),utemp_pp(:,:),diag_precon_pp(:),d_pp(:),         &
     diag_precon_tmp(:,:)
 INTEGER, ALLOCATABLE :: rest(:,:) , g(:), no(:) ,num(:),g_num_pp(:,:),  &
                        g_g_pp(:,:),no_local(:),no_local_temp(:)
!-------------------------input and initialisation---------------------
  timest(1) = elap_time( )   ;  CALL find_pe_procs(numpe,npes)
  IF(numpe==npes) THEN
    OPEN (10,FILE='p122.dat',STATUS=    'OLD',ACTION='READ')
    READ (10,*) phi,c,psi,e,v,cons, nels,nxe,nze,nip,aa,bb,cc,           &
                incs,  plasits,cjits,plastol,cjtol
  END IF
  CALL bcast_inputdata_p122(numpe,npes,phi,c,psi,e,v,cons,nels,nxe,nze,  &
        nip,aa,bb,cc,incs,plasits,cjits,plastol,cjtol)
  CALL calc_nels_pp ; ndof=nod*nodof; nn_temp = 0; neq_temp = 0; ntot=ndof
  nye = nels/nxe/nze; nr = 3*nxe*nye+6*nye*nze+6*nze*nxe+2*nxe+2*nye+1
  nle = nxe/5; loaded_freedoms = 3*nle*nle + 4*nle + 1
  ALLOCATE (rest(nr,nodof+1), points(nip,ndim),weights(nip), m1(nst,nst),&
         num(nod),dee(nst,nst),evpt_pp(nst,nip,nels_pp),  m2(nst,nst),   &
         tensor_pp(nst,nip,nels_pp),coord(nod,ndim),g_g_pp(ntot,nels_pp),&
         stress(nst),storkm_pp(ntot,ntot,nels_pp),jac(ndim,ndim),        &
         der(ndim,nod),deriv(ndim,nod),g_num_pp(nod,nels_pp),            &
         bee(nst,ntot),km(ntot,ntot),eld(ntot),eps(nst),sigma(nst),      &
         bload(ntot),eload(ntot),erate(nst),evp(nst),devp(nst),g(ntot),  &
         m3(nst,nst),flow(nst,nst),pmul_pp(ntot,nels_pp),                &
         utemp_pp(ntot,nels_pp),p_g_co_pp(nod,ndim,nels_pp),             &
         diag_precon_tmp(ntot,nels_pp),no(loaded_freedoms),              &
         no_local_temp(loaded_freedoms),val(loaded_freedoms),qinc(incs))
  CALL loading(nxe,nze,nle,no,val);val=-val*aa*bb/12._iwp;ielpe=iel_start
  IF(numpe==npes) READ(10,*) qinc;   diag_precon_tmp = .0_iwp
  CALL MPI_BCAST(qinc,incs,MPI_REAL8,npes-1,MPI_COMM_WORLD,ier)
  CALL cube_bc20(nxe,nye,nze,rest);  CALL rearrange(rest)
!--------------- loop the elements to set up global arrays --------------
  elements_1: DO iel = 1 , nels_pp
               CALL geometry_20bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
               CALL find_g3(num,g,rest) ; g_num_pp(:,iel) = num
               p_g_co_pp(:,:,iel) = coord; g_g_pp(:,iel)=g; ielpe=ielpe+1
               i = MAXVAL(g);     j = MAXVAL(num)
               IF(i>neq_temp)neq_temp = i; IF(j>nn_temp)nn_temp = j
  END DO elements_1
  neq = reduce(neq_temp); nn = reduce(nn_temp)
   IF(numpe==1) THEN
    OPEN (11,FILE='p122.res',STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I5,A)') "This job ran on ", npes, "  processors"
    WRITE(11,'(A)') "Global coordinates and node numbers "
    DO i= 1, nels_pp, nels_pp - 1
      WRITE(11,'(A,I5)')"Element ",i  ; num = g_num_pp(:,i)
      DO k = 1,nod;WRITE(11,'(A,I5,3E12.4)')                              &
        "  Node",num(k),p_g_co_pp(k,:,i); END DO
    END DO
    WRITE(11,'(A,3(I5,A))') "There are ",nn," nodes",nr,                   &
    " restrained and   ",  neq," equations"
    WRITE(11,*) "Time after setup  is  : ", elap_time( ) - timest(1)
  END IF
  CALL calc_neq_pp  ;  CALL make_ggl(g_g_pp)
 ALLOCATE(loads_pp(neq_pp),bdylds_pp(neq_pp),oldis_pp(neq_pp),           &
  r_pp(neq_pp),totd_pp(neq_pp),p_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp),&
  u_pp(neq_pp),diag_precon_pp(neq_pp),d_pp(neq_pp))
  oldis_pp = .0_iwp; totd_pp=0.0_iwp ; tensor_pp = .0_iwp
  p_pp=.0_iwp; xnew_pp =.0_iwp; diag_precon_pp = .0_iwp
  CALL deemat(dee,e,v); CALL sample(element,points,weights)
  pi = ACOS( -1._iwp ); snph = SIN(phi*pi/180._iwp)
  dt=4._iwp*(1._iwp+v)*(1._iwp-2._iwp*v)/(e*(1._iwp-2._iwp*v+snph*snph))
  IF(numpe==1) WRITE(11,'(A,E12.4)') "The critical timestep is   ",dt
!---- element stiffness integration, preconditioner & set initial stress--
 elements_2: DO iel = 1 , nels_pp
               coord = p_g_co_pp(:,:,iel);g=g_g_pp( : , iel );km=0.0_iwp
            gauss_pts_1:  DO i =1 , nip
              tensor_pp(1:3,i,iel)=cons;  CALL shape_der (der,points,i)
              jac=MATMUL(der,coord);det=determinant(jac);CALL invert(jac)
              deriv = MATMUL(jac,der) ;   CALL beemat (bee,deriv)
              km = km + MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)           &
                             *det*weights(i)
            END DO gauss_pts_1      ;     storkm_pp( :, :, iel) = km
            DO k=1,ntot
              diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)+km(k,k)
            END DO
 END DO elements_2
 CALL scatter(diag_precon_pp,diag_precon_tmp); DEALLOCATE(diag_precon_tmp)
!-------------------- invert preconditioner ----------------------------
    CALL reindex_fixed_nodes                                             &
               (ieq_start,no,no_local_temp,num_no,no_index_start)
    ALLOCATE(no_local(1:num_no)); no_local = no_local_temp(1:num_no)
    DEALLOCATE(no_local_temp)
    diag_precon_pp = 1._iwp / diag_precon_pp
!-----------------------  load  increment loop -------------------------
    load_increments : do iy = 1 , incs
    plasiters=0; bdylds_pp=.0_iwp; evpt_pp=.0_iwp  ;  cjtot = 0
    IF(numpe==npes) WRITE(11,'(/,A,I5)') "Load Increment   ",iy
!-------------------------   plastic iteration loop   ------------------
   plastic_iterations: DO
    plasiters=plasiters+1;  loads_pp=.0_iwp
    DO i=1,num_no
       j =  no_local(i)-ieq_start+1
       loads_pp(j) = val(no_index_start+i-1)*qinc(iy)
    END DO
    loads_pp=loads_pp+bdylds_pp
!------ if x=.0 p and r are just loads but in general p=r=loads-A*x -----
!----------------------so form r = A * x -----------------------------
     r_pp = .0_iwp    ;    CALL gather(x_pp,pmul_pp)
      elements_2a : DO iel = 1 , nels_pp
                utemp_pp(:,iel)=MATMUL(storkm_pp(:,:,iel),pmul_pp(:,iel))
      END DO elements_2a  ;      CALL scatter(r_pp,utemp_pp)
!-----------------------now precondition r and p -----------------------
      r_pp = loads_pp - r_pp; d_pp = diag_precon_pp*r_pp; p_pp = d_pp
```

```
!-------------------   solve the simultaneous equations by pcg -----------
      cjiters = 0
      conjugate_gradients:  DO
       cjiters = cjiters + 1 ; u_pp = .0_iwp   ; pmul_pp = .0_iwp
      CALL gather(p_pp,pmul_pp)
      elements_3 : DO iel = 1 , nels_pp
                 utemp_pp(:,iel)=MATMUL(storkm_pp(:,:,iel),pmul_pp(:,iel))
      END DO elements_3  ;      CALL scatter(u_pp,utemp_pp)
!-------------------------------pcg process ------------------------------
    up =DOT_PRODUCT_P(r_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
    xnew_pp = x_pp + p_pp* alpha; r_pp = r_pp - u_pp*alpha
    d_pp = diag_precon_pp*r_pp ;beta = DOT_PRODUCT_P(r_pp,d_pp)/up
    p_pp = d_pp + p_pp * beta   ;    cj_converged = .TRUE.
    CALL checon_par(xnew_pp,x_pp,cjtol,cj_converged,neq_pp)
    IF(cj_converged.or.cjiters==cjits) EXIT
      END DO conjugate_gradients
      cjtot = cjtot + cjiters
!---------------------------- end of pcg process ------------------------
    loads_pp = xnew_pp     ;  pmul_pp = .0_iwp
!-----------------------   check plastic convergence  -------------------
      CALL checon_par(loads_pp,oldis_pp,plastol,plastic_converged,neq_pp)
      IF(plasiters==1)plastic_converged=.FALSE.
      IF(plastic_converged.OR.plasiters==plasits)bdylds_pp=.0_iwp
      CALL gather(loads_pp,pmul_pp)  ;   utemp_pp = .0_iwp
!------------------------ go round the Gauss Points ---------------------
      elements_4: DO iel = 1 , nels_pp
       bload=.0_iwp; coord =p_g_co_pp( : , :, iel) ; eld = pmul_pp(:,iel)
       gauss_points_2 : DO i = 1 , nip
          CALL shape_der ( der,points,i); jac=MATMUL(der,coord)
          det = determinant(jac);CALL invert(jac); deriv= MATMUL(jac,der)
          CALL beemat (bee,deriv); eps=MATMUL(bee,eld)
          eps = eps - evpt_pp(: ,i ,iel) ; sigma = MATMUL(dee,eps)
          stress = sigma+tensor_pp(: , i, iel)
          CALL invar(stress,sigm,dsbar,lode_theta)
!----------------------  check whether yield is violated  ----------------
         CALL mocouf( phi,c,sigm,dsbar,lode_theta,f)
         IF(plastic_converged.OR.plasiters==plasits) THEN
         devp=stress
           ELSE
           IF(f>=.0) THEN ; CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
           CALL formm(stress,m1,m2,m3) ;  flow=f*(m1*dq1+m2*dq2+m3*dq3)
           erate=MATMUL(flow,stress)  ;  evp=erate*dt
           evpt_pp(:,i,iel)=evpt_pp(:,i,iel)+evp  ;  devp=MATMUL(dee,evp)
         END IF; END IF
      IF(f>=.0) THEN
        eload=MATMUL(TRANSPOSE(bee),devp);bload=bload+eload*det*weights(i)
      END IF
      IF(plastic_converged.OR.plasiters==plasits)THEN
!--------------------- update the Gauss Point stresses  -----------------
          tensor_pp(: , i , iel) = stress
      END IF
    END DO gauss_points_2
!-------------compute the total bodyloads vector ------------------------
         utemp_pp(:,iel) = utemp_pp(:,iel) + bload
  END DO elements_4        ;  CALL scatter(bdylds_pp,utemp_pp)
  IF(plastic_converged.OR.plasiters==plasits)EXIT
 END DO plastic_iterations
 totd_pp=totd_pp+loads_pp
 IF(numpe==1)THEN
 write(11,'(A,E12.4)')"The displacement is  ",totd_pp(1)
 write(11,'(A)')  "  sigma z    sigma x     sigma y "
 write(11,'(3E12.4)')tensor_pp(3,1,1),tensor_pp(1,1,1),tensor_pp(2,1,1)
 write(11,'(A,I12)')"The total number of cj iterations was    ",cjtot
 write(11,'(A,I12)')"The number of plastic iterations was     ",plasiters
 write(11,'(A,F11.2)')"cj iterations per plastic iteration were   ",     &
  & REAL(cjtot)/REAL(plasiters)
 END IF
 IF (plasiters==plasits) EXIT
END DO load_increments
IF(numpe==1) WRITE(11,*) "This analysis took : ", elap_time( ) - timest(1)
CALL shutdown( )
END PROGRAM p122
```

New scalar integers:

`cjiters`	conjugate gradient iteration counter
`cjits`	conjugate gradient iteration ceiling
`cjtot`	total number of conjugate gradient iterations
`incs`	number of load increments
`iy`	simple counter
`plasiters`	plastic iteration counter
`plasits`	plastic iteration ceiling

New scalar reals:

`big`	largest component of a vector
`c`	cohesion
`cjtol`	conjugate gradient iteration tolerance
`cons`	consolidation pressure
`dq1`	Mohr–Coulomb plastic potential derivative
`dq2`	Mohr–Coulomb plastic potential derivative
`dq3`	Mohr–Coulomb plastic potential derivative
`dsbar`	shear stress invariant
`dt`	viscoplastic “time” step
`f`	current stress state
`lode_theta`	Lode angle
`phi`	angle of internal friction
`plastol`	plastic iteration tolerance
`presc`	prescribed value of load/displacement
`psi`	angle of dilation
`sigm`	mean stress invariant
`sinph`	sine of angle of internal friction

New scalar logicals:

`cj_converged`	set to `.TRUE.` if conjugate gradient iterations converged
`plastic_converged`	set to `.TRUE.` if plastic iterations converged

New dynamic real arrays:

`bdylds_pp`	distributed body loads vector
`bload`	element body loads
`devp`	increment of viscoplastic strain
`eload`	accumulating element body loads
`erate`	viscoplastic strain rate
`evp`	viscoplastic strains
`evpt_pp`	distributed total viscoplastic strains
`flow`	viscoplastic flow
`loads_pp`	distributed loads vector
`m1`	plastic potential derivatives matrix
`m2`	plastic potential derivatives matrix
`m3`	plastic potential derivatives matrix
`oldis_pp`	previous distributed displacements
`qinc`	vector of load increment terms
`storkm_pp`	distributed stored element stiffness matrices
`stress`	stress vector
`tensor_pp`	distributed stresses
`totd_pp`	distributed total displacements
`xnew_pp`	updated distributed pcg vector

This program follows naturally from the previous one by extending the material behaviour beyond the elastic range. The post elastic region is perfectly plastic governed by the Mohr–Coulomb yield criterion that was first introduced in Program 6.3 and applied in three-dimensional analyses in Program 6.11. The same geometric restrictions apply as in the previous program-only cuboidal meshes are allowed with the central loaded patch a vertical uniformly distributed load extending to one-fifth of the surface in the x and y directions (z is vertical). Thus only multiples of 5 should be used for `nxe` in this simple case.

After `INTEGER` and `REAL` data have been read and broadcast, load increment array `qinc` (See Programs 6.1 and 6.2) is read and needs to be broadcast to all other processors using `MPI_BCAST`. The loop labelled `elements_1` can be compared with the same loop in Program 6.11. In parallel, `geometry_20bxz` is used in place of the serial `geom_rect` but the basic process of finding `coord`, `num` and `g` for each element is clearly equivalent.

Similarly loop labelled `elements_2` with its embedded integration loop `gauss_pts_1` carries over essentially unaltered from serial to parallel versions.

In Program 6.11 each set of conjugate gradient iterations begins with a starting `x` value of zero. In large non-linear problems it is beneficial to use the value of `x` computed in the previous step as the starting `x` for a new step. Typically this halves the number of pcg iterations in subsequent steps, and this procedure is used in this program (see loop `elements_2a`).

Loops labelled `conjugate_gradients`, `elements_3`, and `elements_4` with its embedded `gauss_points_2` are equivalent in serial and parallelised versions.

The problem analysed as shown in Figure 12.1 with data in Figure 12.14, deals with the bearing capacity of a square uniformly loaded punch at the surface of an elasto-plastic block. Since ϕ (`phi`) is set to zero, the Mohr–Coulomb criterion reduces to the Tresca, and starting stresses can be set to zero. There is no analytical solution to this problem, but approximate solutions indicate an increase in 3D above the $(2 + \pi)c_u$ recorded in plane strain.

In our case the 3D ultimate load is computed to be about $5.4c_u$, as shown in the results listed in Figure 12.15 Performance data are shown in Figure 12.16

```
phi   c       psi   e       v     cons
0.0  100.0    0.0  100.0   0.3    0.0

nels   nxe   nze   nip
1000    10    10    8

aa   bb   cc   incs
1.0  1.0  1.0   12

plasits    cjits   plastol  cjtol
250         1000    1.0e-4  1.0e-5

incs,(qinc(i),i=1,incs)
200.0  100.0  50.0  50.0  50.0  30.0
 20.0   10.0  10.0   5.0   5.0   5.0
```

Figure 12.14 Data for Program 12.2 example

```
This job ran on     8  processors

There are  4961 nodes 1541 restrained and   12580 equations
 Time after setup  is  :  0.169999999998253770
The critical timestep is     0.5200E-01
The displacement is   -0.6831E+01
  sigma z    sigma x     sigma y
 -0.1966E+03 -0.1236E+03 -0.1236E+03
The total number of cj iterations was            157
The number of plastic iterations was               2
cj iterations per plastic iteration were        78.50
The displacement is   -0.1033E+02
  sigma z    sigma x     sigma y
 -0.2853E+03 -0.1808E+03 -0.1808E+03
The total number of cj iterations was            388
The number of plastic iterations was               7
cj iterations per plastic iteration were        55.43
.
.
.
The displacement is   -0.4098E+02
  sigma z    sigma x     sigma y
 -0.5375E+03 -0.3586E+03 -0.3586E+03
The total number of cj iterations was           6040
The number of plastic iterations was             138
cj iterations per plastic iteration were        43.77
 This analysis took :  593.080000000001746
```

Figure 12.15 Results from Program 12.2 example

Mesh	No of Processors	Analysis Time(secs)
10x10x10	8	593
	16	375

Figure 12.16 Performance statistics: Program 12.2 (IBM SP2)

Program 12.3 Three dimensional Laplacian flow. Compare Program 7.5.

```
PROGRAM p123
!-----------------------------------------------------------------------
!    Program 7.5 three dimensional analysis of Laplace's equation
!    using 8-node brick elements, preconditioned conjugate gradient solver
!    only integrate one element , diagonal preconditioner diag_precon
!    parallel version  ;   central loaded or fixed freedom   ;  box_bc
!-----------------------------------------------------------------------
  USE new_library; USE geometry_lib; USE precision; USE utility
  USE mp_module  ;  USE  timing    ; USE global_variables1
  USE gather_scatter6;  IMPLICIT NONE
 !  ndof, nels, neq , ntot  are now global variables - not declared
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=1,nod=8, nres, is , it ,           &
          i,j,k,ndim=3,iters,limit,iel,num_no,no_index_start,            &
          neq_temp,nn_temp , loaded_freedoms, fixed_freedoms
 REAL(iwp)::aa,bb,cc,kx,ky,kz,det,tol,up,alpha,beta,q,penalty=1.e20_iwp
 CHARACTER(LEN=15)::element= 'hexahedron';    LOGICAL :: converged
!------------------------- dynamic arrays-------------------------------
 REAL(iwp),ALLOCATABLE :: points(:,:),kc(:,:),coord(:,:), weights(:),     &
                        p_g_co_pp(:,:,:), jac(:,:), der(:,:), deriv(:,:),&
                        col(:,:),row(:,:),kcx(:,:),kcy(:,:),kcz(:,:),    &
                        diag_precon_pp(:),p_pp(:),r_pp(:),x_pp(:),       &
                        xnew_pp(:),u_pp(:),pmul_pp(:,:),utemp_pp(:,:),   &
                        d_pp(:),diag_precon_tmp(:,:),val(:),val_f(:),    &
                        store_pp(:),eld(:)
 INTEGER, ALLOCATABLE :: rest(:,:),g(:),num(:),g_num_pp(:,:),g_g_pp(:,:), &
          no(:),no_f(:),no_local_temp(:),no_local_temp_f(:),no_local(:)
!------------------------input and initialisation------------------------
 timest(1) = elap_time( )   ;    CALL find_pe_procs(numpe,npes)
 IF (numpe==npes) THEN
  OPEN (10,FILE='p123.dat',STATUS=    'OLD',ACTION='READ')
  READ (10,*) nels,nxe,nze,nip,aa,bb,cc,kx,ky,kz, tol,limit ,            &
              loaded_freedoms,fixed_freedoms
 END IF
 CALL bcast_inputdata_p123(numpe,npes,nels,nxe,nze,nip,aa,bb,cc,kx,ky,kz, &
                           tol,limit,loaded_freedoms,fixed_freedoms)
 CALL calc_nels_pp   ;   ndof=nod*nodof ; ntot=ndof ; nye = nels/nxe/nze
      neq_temp = 0; nn_temp = 0 ; nr=(nxe+1)*(nye+1)+(nxe+1)*nze+nye*nze
 ALLOCATE (points(nip,ndim),coord(nod,ndim),jac(ndim,ndim),kc(ntot,ntot), &
        der(ndim,nod),deriv(ndim,nod),rest(nr,nodof+1),kcx(ntot,ntot),    &
        g(ntot),pmul_pp(ntot,nels_pp),utemp_pp(ntot,nels_pp),col(ntot,1), &
        p_g_co_pp(nod,ndim,nels_pp),g_num_pp(nod,nels_pp),weights(nip),   &
        num(nod),g_g_pp(ntot,nels_pp),no(1),kcy(ntot,ntot),val(1),        &
        no_local_temp(1),row(1,ntot),diag_precon_tmp(ntot,nels_pp),       &
        val_f(1),eld(ntot),no_f(1),no_local_temp_f(1),kcz(ntot,ntot))
      CALL box_bc8(nxe,nye,nze,rest); ielpe=iel_start; nres=nxe*(nze-1)+1
      IF(loaded_freedoms>0) THEN; no = nres;   val = 10._iwp; END IF
      IF(fixed_freedoms>0)THEN; no_f=nres; val_f = 100._iwp ; END IF
      CALL sample(element,points,weights); CALL rearrange_2(rest)
!---------- loop the elements for global cordinates etc -------------------
     elements_0: DO iel = 1 , nels_pp
             CALL geometry_8bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
             CALL find_g4(num,g,rest) ; g_num_pp(:,iel) = num
             p_g_co_pp(:,:,iel) = coord; g_g_pp(:,iel)=g; ielpe = ielpe+1
             i = MAXVAL(g); j = MAXVAL(num)
             IF(i>neq_temp)neq_temp = i; IF(j>nn_temp)nn_temp = j
     END DO elements_0
```

```
   neq = reduce(neq_temp)  ;  nn = reduce(nn_temp)
   CALL calc_neq_pp    ;  CALL make_ggl(g_g_pp); diag_precon_tmp = .0_iwp
   DO i=1,neq_pp; IF(nres==ieq_start+i-1) THEN; it = numpe;is = i; END IF
   END DO
   ALLOCATE(p_pp(neq_pp),r_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp),       &
        u_pp(neq_pp),diag_precon_pp(neq_pp),d_pp(neq_pp),store_pp(neq_pp))
        r_pp = .0_iwp; p_pp = .0_iwp; x_pp = .0_iwp; xnew_pp = .0_iwp
        diag_precon_pp = .0_iwp ;   store_pp = .0_iwp
!------- element stiffness integration and build the preconditioner-------
      iel=1;CALL geometry_8bxz(iel,nxe,nze,aa,bb,cc,coord,num)
      kcx = .0_iwp; kcy = .0_iwp; kcz = .0_iwp
      gauss_pts_1:  DO i=1,nip
             CALL shape_der (der,points,i) ; jac = MATMUL(der,coord)
             det=determinant(jac);CALL invert(jac);deriv = MATMUL(jac,der)
             row(1,:) = deriv(1,:); eld=deriv(1,:); col(:,1) = eld
             kcx = kcx + MATMUL(col,row)*det*weights(i)
             row(1,:) = deriv(2,:); eld=deriv(2,:); col(:,1) = eld
             kcy = kcy + MATMUL(col,row)*det*weights(i)
             row(1,:) = deriv(3,:); eld=deriv(3,:); col(:,1) = eld
             kcz = kcz + MATMUL(col,row)*det*weights(i)
      END DO gauss_pts_1        ; kc = kcx*kx + kcy*ky + kcz*kz
     elements_1: DO iel = 1,nels_pp
       DO k=1,ntot
         diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)+kc(k,k);END DO
     END DO elements_1
  CALL scatter(diag_precon_pp,diag_precon_tmp);DEALLOCATE(diag_precon_tmp)
     IF(numpe==it)THEN
      OPEN (11,FILE='p123.res',STATUS='REPLACE',ACTION='WRITE')
      WRITE(11,'(A,I5,A)') "This job ran on ", npes , "  processors"
      WRITE(11,'(A)') "Global coordinates and node numbers "
      DO i = 1 , nels_pp,nels_pp-1
                 WRITE(11,'(A,I8)')"Element ",i ; num = g_num_pp(:,i)
      DO k = 1,nod;WRITE(11,'(A,I8,3E12.4)')                              &
                 "  Node",num(k),p_g_co_pp(k,:,i); END DO
      END DO
      WRITE(11,'(A,3(I8,A))') "There are ",nn," nodes",nr,                &
                           " restrained and", neq," equations"
      WRITE(11,*) "Time after setup is   :", elap_time( ) - timest(1)
     END IF
!-------------------- get starting r-------------------------------------
    IF(loaded_freedoms>0) THEN
     CALL reindex_fixed_nodes                                              &
                 (ieq_start,no,no_local_temp,num_no,no_index_start)
     ALLOCATE(no_local(1:num_no)) ; no_local = no_local_temp(1:num_no)
     DEALLOCATE(no_local_temp)
        DO i = 1 , num_no
         r_pp(no_local(i)-ieq_start+1) = val(no_index_start + i - 1)
        END DO
    END IF            ;     q = SUM_P(r_pp)
    IF(numpe==it)THEN  ;  WRITE(11,'(A,E12.4)') "The total load is ", q
    END IF
    IF(fixed_freedoms>0) THEN
     CALL reindex_fixed_nodes(ieq_start,no_f,no_local_temp_f,              &
                              num_no,no_index_start)
     ALLOCATE(no_local(1:num_no)) ; no_local = no_local_temp_f(1:num_no)
     DEALLOCATE(no_local_temp_f)
        DO i = 1 , num_no        ; j=no_local(i) - ieq_start + 1
         diag_precon_pp(j)=diag_precon_pp(j) + penalty
         r_pp(j) = diag_precon_pp(j) *  val_f(no_index_start + i - 1)
         store_pp(j) =  diag_precon_pp(j)
        END DO
    END IF
    diag_precon_pp=1._iwp/diag_precon_pp;d_pp=diag_precon_pp*r_pp;p_pp=d_pp
!--------------------preconditioned c. g. iterations-----------------------
      iters = 0
     iterations  :      DO
             iters = iters + 1     ;    u_pp = 0._iwp  ; pmul_pp = .0_iwp
       CALL gather(p_pp,pmul_pp)
       elements_2 : DO iel = 1, nels_pp
                    utemp_pp(:,iel) = MATMUL(kc,pmul_pp(:,iel))
       END DO elements_2  ;       CALL scatter(u_pp,utemp_pp)
    IF(fixed_freedoms>0) THEN
        DO i = 1 , num_no;  j = no_local(i)-ieq_start+1
         u_pp(j)=p_pp(j) * store_pp(j)
        END DO
    END IF
!--------------------------pcg equation solution--------------------------
          up=DOT_PRODUCT_P(r_pp,d_pp); alpha= up/ DOT_PRODUCT_P(p_pp,u_pp)
          xnew_pp = x_pp + p_pp* alpha ; r_pp=r_pp - u_pp*alpha
          d_pp = diag_precon_pp*r_pp ;  beta=DOT_PRODUCT_P(r_pp,d_pp)/up
          p_pp=d_pp+p_pp*beta
          CALL checon_par(xnew_pp,x_pp,tol,converged,neq_pp)
          IF(converged .OR. iters==limit) EXIT
     END DO iterations
     IF(numpe==it)THEN
       WRITE(11,'(A,I5)')"The number of iterations to convergence was  ", &
                          iters
       WRITE(11,'(A)')   "The  potentials are   :"
       WRITE(11,'(A)') "   Freedom       Potential"
       DO i = 1 , 4
        WRITE(11,'(I5,A,E12.4)') nres+i-1, "     ", xnew_pp(is+i-1)
       END DO
     END IF
  IF(numpe==it) WRITE(11,*)"This analysis took   ", elap_time( )-timest(1)
  CALL shutdown( )
 END PROGRAM p123
```

New scalar integers:

`fixed_freedoms`	number of fixed freedoms
`is`	location of desired output freedom
`it`	processor on which desired output resides
`nres`	number of output freedom

New scalar reals:

`kx`	conductivity in x-direction
`ky`	conductivity in y-direction
`kz`	conductivity in z-direction
`penalty`	value of penalty restraint

New dynamic integer arrays:

`no_f`	vector of fixed freedom numbers
`no_local_temp_f`	temporary vector

New dynamic real arrays:

`col`	column array
`kc`	conductivity matrix
`kcx`	x contribution to conductivity matrix
`kcy`	y contribution to conductivity matrix
`kcz`	z contribution to conductivity matrix
`row`	row array
`store_pp`	distributed penalty storage
`val_f`	values of fixed freedoms

The closest serial equivalent to this program is Program 7.5. Geometrical restrictions are to a quarter of a cuboidal box with zero potential on all its outer faces. A potential of 100.0 units can be fixed at the centre of the box or a flux of 10.0 units applied there, so either `loaded_freedoms` or `fixed_freedoms` must be set to 1 and the other to 0.

Comparison of Programs 7.5 and 12.1 will show the familiar patterns of analysis type and parallelisation process respectively. In the serial programs, loop `elements_1` contains integration loop `gauss_pts_1` embedded within it whereas in the parallel version these are separate as `elements_0` and `gauss_pts_1`. An extra loop `elements_1` is necessary for formation of the preconditioner. For simplicity, in parallel a single `kc` matrix is used. Loop `elements_2` is equivalent in both versions but in parallel, flow rates are

```
nels      nxe   nze   nip
1000000   100   100    8

aa     bb     cc
0.01   0.01   0.01

kx    ky    kz
2.0   2.0   2.0


tol      limit
1.0e-5   500


loaded_freedoms   fixed_freedoms
1                       0
```

Figure 12.17 Data for Program 12.3 example

```
This job ran on     8  processors

There are  1030301 nodes   30301 restrained and 1000000 equations
 Time after setup is   : 2.37000000000261934
The total load is   0.1000E+02
The number of iterations to convergence was    121
The  potentials are :
   Freedom       Potential
 9901       0.1748E+04
 9902       0.1713E+03
 9903       0.1586E+03
 9904       0.9910E+02
 This analysis took    382.870000000002619
```

Figure 12.18 Results from Program 12.3 example

Mesh	No of Processors	Analysis Time(secs)
100x100x100	8	383
	16	190

Number of equations	Iterations to convergence
125,000	72
1,000,000	121
8,000,000	180

Figure 12.19 Performance statistics: Program 12.3 (IBM SP2)

not retrieved at the end of the analysis. The appropriate freedom for output is identified as `is` and the processor on which it resides as `it`.

The problem analysed with the data shown in Figure 12.17 is of a quarter cube of unit size with a flux of 10.0 units applied at the centre. One million equations are involved in this analysis. Results are listed as Figure 12.18 and illustrations of increasing pcg iteration counts with problem size, together with speedup are shown in Figure 12.19.

Program 12.4 Three dimensional transient flow- implicit analysis in time. Compare Program 8.3.

```
  PROGRAM p124
!-----------------------------------------------------------------------
!      Program 8.3 conduction equation on 3-d box shaped
!      volume using 8-node hexahedral elements : pcg version implicit
!      integration in time using 'theta' method : parallel version : box_bc
!-----------------------------------------------------------------------
 USE new_library; USE geometry_lib; USE precision; USE utility
 USE mp_module  ; USE timing    ;  USE global_variables1
 USE gather_scatter6;  IMPLICIT NONE
! ndof,nels,neq,ntot are now global variables - not declared
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=1,nod=8,ndim=3,neq_temp,nn_temp,    &
          i,j,k,iel,nstep,npri,nres,iters,limit, it,is
 REAL(iwp)::aa,bb,cc,kx,ky,kz,det,theta,dtim,val0,real_time,              &
            tol,alpha,beta,up,big
 LOGICAL::converged       ; CHARACTER(LEN=15) :: element='hexahedron'
!------------------------ dynamic arrays---------------------------------
 REAL(iwp),ALLOCATABLE ::loads_pp(:),u_pp(:),p_pp(:),points(:,:),kay(:,:),&
                       coord(:,:),fun(:),jac(:,:),der(:,:),deriv(:,:),     &
                       weights(:),d_pp(:),kc(:,:), pm(:,:), funny(:,:),    &
                       p_g_co_pp(:,:,:),storka_pp(:,:,:),storkb_pp(:,:,:), &
                       x_pp(:),xnew_pp(:),pmul_pp(:,:),utemp_pp(:,:),      &
                       diag_precon_pp(:),diag_precon_tmp(:,:)
 INTEGER, ALLOCATABLE :: rest(:,:), g(:), num(:),g_num_pp(:,:),g_g_pp(:,:)
!---------------------input and initialisation--------------------------
 timest(1) = elap_time( )   ; CALL find_pe_procs(numpe,npes)
 IF(numpe==npes) THEN
  OPEN (10,FILE='p124.dat',STATUS=    'OLD',ACTION='READ')
  READ (10,*) nels,nxe,nze,nip,aa,bb,cc,kx,ky,kz,                         &
            dtim,nstep,theta,npri,tol,limit, val0
 END IF
 CALL bcast_inputdata_p124(numpe,npes,nels,nxe,nze,nip,aa,bb,cc,kx,       &
                 ky,kz,dtim,nstep,theta,npri,tol,limit,val0)
 CALL calc_nels_pp; ndof=nod*nodof; ntot = ndof; nn_temp = 0; neq_temp = 0
 nye=nels/nxe/nze; nr=(nxe+1)*(nye+1)+(nxe+1)*nze+nye*nze; ielpe=iel_start
  ALLOCATE (rest(nr,nodof+1),points(nip,ndim),weights(nip),kay(ndim,ndim),&
            coord(nod,ndim),fun(nod),jac(ndim,ndim),der(ndim,nod),g(ntot),&
            p_g_co_pp(nod,ndim,nels_pp),deriv(ndim,nod), pm(ntot,ntot),   &
            g_num_pp(nod,nels_pp),kc(ntot,ntot),funny(1,nod),num(nod),    &
            g_g_pp(ntot,nels_pp),storka_pp(ntot,ntot,nels_pp),            &
            utemp_pp(ntot,nels_pp),storkb_pp(ntot,ntot,nels_pp),          &
            pmul_pp(ntot,nels_pp),diag_precon_tmp(ntot,nels_pp))
  kay=0.0_iwp ; kay(1,1) = kx; kay(2,2) = ky ; kay(3,3) = kz
  CALL sample (element,points,weights)  ; CALL box_bc8(nxe,nye,nze,rest)
  CALL rearrange_2(rest)
!-------------loop the elements to  set up global arrays -----------------
    elements_1: DO iel = 1 , nels_pp
               CALL geometry_8bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
               CALL find_g4(num,g,rest) ; g_num_pp(:,iel) = num
               p_g_co_pp(:,:,iel) = coord; g_g_pp(:,iel)=g; ielpe=ielpe+1
               i = MAXVAL(g);   j = MAXVAL(num)
               IF(i>neq_temp)neq_temp = i; IF(j>nn_temp)nn_temp = j
    END DO elements_1
    neq = reduce(neq_temp) ; nn = reduce(nn_temp) ; nres = nxe*(nze-1) + 1
    CALL calc_neq_pp ; CALL make_ggl(g_g_pp)
    DO i = 1, neq_pp; IF(nres==ieq_start+i-1) THEN;it=numpe;is = i; END IF
    END DO
    IF(numpe==it) THEN
     OPEN (11,FILE='p124.res',STATUS='REPLACE',ACTION='WRITE')
     WRITE(11,'(A,I5,A)') "This job ran on  ", npes, "  processors"
     WRITE(11,'(A)') "Global coordinates and node numbers "
     DO i= 1, nels_pp , nels_pp - 1
           WRITE(11,'(A,I8)')"Element ",i  ; num = g_num_pp(:,i)
      DO k = 1,nod;WRITE(11,'(A,I8,3E12.4)')                              &
        "  Node",num(k),p_g_co_pp(k,:,i); END DO
     END DO
     WRITE(11,'(A,3(I8,A))')"There are ",nn," nodes",nr," restrained and",&
                           neq," equations"
     WRITE(11,*) "Time after setup  is   :", elap_time( ) - timest(1)
    END IF
    ALLOCATE(loads_pp(neq_pp),diag_precon_pp(neq_pp),u_pp(neq_pp),        &
             d_pp(neq_pp),p_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp))
      storka_pp = .0_iwp; storkb_pp = .0_iwp;diag_precon_tmp = .0_iwp
      p_pp = .0_iwp; diag_precon_pp = .0_iwp; xnew_pp = .0_iwp
!----------- element integration ,storage and build preconditioner --------
    elements_2: DO iel = 1 , nels_pp
                 num = g_num_pp( : , iel); g =  g_g_pp( :, iel)
                 coord = p_g_co_pp( : , : , iel) ; kc=0.0_iwp ; pm=0.0_iwp
       gauss_pts:  DO i =1 , nip
               CALL shape_der (der,points,i); CALL shape_fun(fun,points,i)
               funny(1,:)=fun(:) ; jac = MATMUL(der,coord)
               det=determinant(jac);CALL invert(jac);deriv=MATMUL(jac,der)
               kc = kc + MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)       &
                    *det*weights(i)
               pm  =  pm + MATMUL(TRANSPOSE(funny),funny)*det*weights(i)
       END DO gauss_pts
     storka_pp(:,:,iel)=pm+kc*theta*dtim
     storkb_pp(:,:,iel)=pm-kc*(1._iwp-theta)*dtim
     DO k=1,ntot
      diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)+ storka_pp(k,k,iel)
     END DO
    END DO elements_2
  CALL scatter(diag_precon_pp,diag_precon_tmp);DEALLOCATE(diag_precon_tmp)
    diag_precon_pp = 1._iwp/diag_precon_pp
!-----------------------------initial conditions ------------------------------
    loads_pp=val0         ;    pmul_pp = .0_iwp
!----------------------time stepping recursion ------------------------------
 IF(numpe==it) THEN
   WRITE(11,'(A)') "    Time     Pressure  Iterations"
 END IF
  timesteps: DO j=1,nstep
              real_time=j*dtim    ;      u_pp = .0_iwp
    CALL gather(loads_pp,pmul_pp)
    elements_3 : DO iel = 1 , nels_pp
             utemp_pp(:,iel) =  MATMUL(storkb_pp(:,:,iel),pmul_pp(:,iel))
    END DO elements_3 ;    CALL scatter(u_pp,utemp_pp) ;  loads_pp = u_pp
!------------------- solve simultaneous equations by pcg ------------------
       d_pp = diag_precon_pp*loads_pp; p_pp = d_pp; x_pp = .0_iwp
       iters = 0
     iterations  :      DO
             iters = iters + 1      ;    u_pp = 0._iwp  ; pmul_pp = .0_iwp
    CALL gather(p_pp,pmul_pp)
    elements_4 :  DO iel = 1, nels_pp
          utemp_pp(:,iel) = MATMUL(storka_pp(:,:,iel),pmul_pp(:,iel))
    END DO elements_4;    CALL scatter(u_pp,utemp_pp)
!--------------------------pcg equation solution---------------------------
        up=DOT_PRODUCT_P(loads_pp,d_pp);alpha= up/DOT_PRODUCT_P(p_pp,u_pp)
        xnew_pp = x_pp + p_pp* alpha ; loads_pp=loads_pp - u_pp*alpha
        d_pp=diag_precon_pp*loads_pp;beta=DOT_PRODUCT_P(loads_pp,d_pp)/up
        p_pp=d_pp+p_pp*beta    ;           u_pp = xnew_pp
        CALL checon_par(xnew_pp,x_pp,tol,converged,neq_pp)
        IF(converged .OR. iters==limit) EXIT
     END DO iterations
        loads_pp=xnew_pp
        IF(j/npri*npri==j .AND. numpe == it)  WRITE(11,'(2E12.4,I5)')     &
                            real_time,loads_pp(is),iters
  END DO timesteps
 IF(numpe==it) WRITE(11,*) "This analysis took  :" ,elap_time( )-timest(1)
 CALL shutdown( )
END PROGRAM p124
```

New scalar integers:

`npri`	print interval
`nstep`	number of timesteps in analysis

New scalar reals:

`dtim`	timestep
`real_time`	accumulated time
`theta`	parameter in “theta” integrator
`val0`	initial value

New dynamic real arrays:

`fun`	element shape functions
`funny`	intermediate array
`kay`	conductivity matrix
`pm`	element mass matrix

`storka_pp`	distributed storage of pm and km
`storkb_pp`	distributed storage of pm and km

The closest serial equivalent is Program 8.3. As in the previous program, geometry restrictions are to a symmetrical quarter cuboidal box mesh which has zero potential on all external faces. An initial condition of uniform potential `val0` is specified everywhere else, and the decay of potential at the centre of the box with time is monitored.

Loops `elements_1` and `element_2` with its embedded `gauss_pts` can be traced clearly between the serial and parallelised versions, as can loops `elements_3` and `elements_4`.

The problem analysed as shown in Figure 12.2 is for a quarter cube of unit size with an initial potential of 100.0 units everywhere except at the external boundaries. Data are listed as Figure 12.20 with results as Figure 12.21 and performance statistics as Figure 12.22

```
nels    nxe  nze  nip
125000   50   50   8

aa     bb     cc     kx   ky   kz
0.02  0.02  0.02  1.0  1.0  1.0

dtim  nsteps  theta
0.01   150     0.5

npri  tol     limit   val0
10   0.0001   100    100.0
```

Figure 12.20 Data for Program 12.4 example

```
This job ran on     16  processors

There are   132651 nodes    7651 restrained and  125000 equations
 Time after setup  is   : 0.510000000002037268
    Time     Pressure  Iterations
  0.1000E+00  0.8564E+02   21
  0.2000E+00  0.4605E+02   19
  0.3000E+00  0.2229E+02   14
  0.4000E+00  0.1066E+02   13
  0.5000E+00  0.5085E+01   12
.
.
.
  0.1300E+01  0.1360E-01   10
  0.1400E+01  0.6484E-02   10
  0.1500E+01  0.3087E-02   10
 This analysis took  : 421.750000000000000
```

Figure 12.21 Results from Program 12.3 example

Mesh	No of Processors	Analysis Time(secs)
50x50x50	8	840
	16	422

Figure 12.22 Performance statistics: Program 12.4 (IBM SP2)

Program 12.5 Three dimensional transient flow-explicit analysis in time. Compare Program 8.4.

```
  PROGRAM p125
!-------------------------------------------------------------------------
!   Program 8.4 conduction equation on a 3-d box volume using 8-node
!   hexahedral elements and a simple explicit algorithm : parallel version
!   box_bc ; write on processor it at freedom nres
!-------------------------------------------------------------------------
 USE new_library; USE geometry_lib; USE precision; USE utility
 USE mp_module  ; USE  timing    ; USE global_variables1
 USE gather_scatter6;  IMPLICIT NONE
!  ndof, nels, neq , ntot  are now global variables - not declared
  INTEGER::nxe,nye,nze,nn,nr,nip,nodof=1,nod=8,ndim=3,i,j,k,iel,         &
           neq_temp,nn_temp,nstep,npri,nres   , it,is
  REAL(iwp)::aa,bb,cc,kx,ky,kz,det,dtim,val0,real_time
  CHARACTER (LEN=15) :: element = 'hexahedron'
!------------------------ dynamic arrays---------------------------------
 REAL(iwp),ALLOCATABLE ::loads_pp(:),points(:,:),kay(:,:),coord(:,:),     &
                        jac(:,:),der(:,:),deriv(:,:),weights(:),kc(:,:),    &
                        pm(:,:), funny(:,:),p_g_co_pp(:,:,:),globma_pp(:),  &
                        fun(:),store_pm_pp(:,:,:), newlo_pp(:),mass(:),     &
                        globma_tmp(:,:),pmul_pp(:,:),utemp_pp(:,:)
 INTEGER, ALLOCATABLE :: rest(:,:), g(:),num(:), g_num_pp(:,:),g_g_pp(:,:)
!----------------------input and initialisation--------------------------
  timest(1) = elap_time( )  ;  CALL find_pe_procs(numpe,npes)
  IF(numpe==npes) THEN
   OPEN (10,FILE='p125.dat',STATUS=    'OLD',ACTION='READ')
   READ (10,*) nels,nxe,nze,nip,aa,bb,cc,kx,ky,kz,dtim,nstep,npri,val0
  END IF
  CALL bcast_inputdata_p125(numpe,npes,nels,nxe,nze,nip,aa,bb,cc,         &
                            kx,ky,kz,dtim,nstep,npri,val0)
  CALL calc_nels_pp; ndof=nod*nodof; nn_temp=0;  neq_temp = 0; ntot = ndof
  nye = nels/nxe/nze; nr = (nxe+1)*(nye+1) + (nxe+1)*nze + nye*nze
  ALLOCATE (rest(nr,nodof+1),points(nip,ndim),weights(nip),kay(ndim,ndim),&
         coord(nod,ndim),jac(ndim,ndim),p_g_co_pp(nod,ndim,nels_pp),      &
         der(ndim,nod),deriv(ndim,nod),g_num_pp(nod,nels_pp),             &
         kc(ntot,ntot),g(ntot),funny(1,nod),num(nod),g_g_pp(ntot,nels_pp),&
         store_pm_pp(ntot,ntot,nels_pp),mass(ntot),fun(nod),pm(ntot,ntot),&
         globma_tmp(ntot,nels_pp),pmul_pp(ntot,nels_pp),                  &
         utemp_pp(ntot,nels_pp))
  kay=0.0_iwp; kay(1,1) = kx ; kay(2,2) = ky; kay(3,3) = kz
  ielpe=iel_start ; CALL box_bc8(nxe,nye,nze,rest); CALL rearrange_2(rest)
!---------------loop the elements to  set up global arrays ----------------
   elements_0: DO iel = 1 , nels_pp
               CALL geometry_8bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
               CALL find_g4(num,g,rest) ; g_num_pp(:,iel) = num
               p_g_co_pp(:,:,iel) = coord; g_g_pp(:,iel)=g; ielpe=ielpe+1
               i = MAXVAL(g);   j = MAXVAL(num)
               IF(i>neq_temp)neq_temp = i  ; IF(j>nn_temp)nn_temp = j
   END DO elements_0
   neq=reduce(neq_temp)  ;  nn = reduce(nn_temp); nres = nxe*(nze-1) + 1
   CALL calc_neq_pp     ;      CALL make_ggl(g_g_pp)
   DO i=1,neq_pp; IF(nres==ieq_start+i-1) THEN;it=numpe;is=i;END IF;END DO
   IF(numpe==it) THEN
    OPEN (11,FILE='p125.res',STATUS= 'REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I5,A)') "This job ran on " , npes , "  processors"
    WRITE(11,'(A)') "Global coordinates and node numbers "
    DO i= 1, nels_pp,nels_pp-1 ; WRITE(11,'(A,I8)')"Element ",i
```

```
      num = g_num_pp(:,i)
      DO k = 1,nod;WRITE(11,'(A,I8,3E12.4)')                             &
        "  Node",num(k),p_g_co_pp(k,:,i); END DO
    END DO
    WRITE(11,'(A,3(I8,A))') "There are ",nn," nodes",nr," restrained and",&
                           neq," equations"
    WRITE(11,*) "Time after setup  is  :" , elap_time( ) - timest(1)
   END IF
    CALL sample(element,points,weights) ; globma_tmp = .0_iwp
    ALLOCATE(loads_pp(neq_pp),newlo_pp(neq_pp),globma_pp(neq_pp))
    loads_pp=.0_iwp; newlo_pp=.0_iwp; globma_pp=.0_iwp
!------------ loop the elements for integration and invert mass -----------
 elements_1: DO iel = 1 , nels_pp
              coord = p_g_co_pp(:,:,iel);  kc=0.0_iwp ; pm=0.0_iwp
       gauss_pts:  DO i =1 , nip
           CALL shape_der (der,points,i); CALL shape_fun(fun,points,i)
           funny(1,:)=fun(:) ; jac = MATMUL(der,coord)
           det=determinant(jac); CALL invert(jac); deriv = MATMUL(jac,der)
           kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
           pm  =  pm + MATMUL(TRANSPOSE(funny),funny)*det*weights(i)
       END DO gauss_pts
      DO i=1,ntot; mass(i) = sum(pm(i,:)); END DO
      pm = .0_iwp ; DO i = 1 , ntot; pm(i,i) = mass(i); END DO
      store_pm_pp(:,:,iel) = pm - kc*dtim
      DO i=1,ntot; globma_tmp(i,iel)=globma_tmp(i,iel)+mass(i); END DO
 END DO elements_1
 IF(numpe==it)                                &
  WRITE(11,*) "Time after element integration is :",elap_time( )-timest(1)
 CALL scatter(globma_pp,globma_tmp)
 globma_pp = 1._iwp/globma_pp  ; loads_pp = val0 ; DEALLOCATE(globma_tmp)
!------------------time stepping recursion--------------------------------
  IF(numpe==it) THEN
   WRITE(11,'(A)') "    Time     Pressure"
  END IF
  timesteps: DO j=1,nstep
              real_time=j*dtim
!---------------  go round the elements  ---------------------------------
    pmul_pp = .0_iwp ; CALL gather(loads_pp,pmul_pp)  ;  utemp_pp = .0_iwp
    elements_2 : DO iel = 1 , nels_pp
                 pm = store_pm_pp(: , : , iel)
                 utemp_pp(:,iel)=utemp_pp(:,iel)+MATMUL(pm,pmul_pp(:,iel))
    END DO elements_2
    CALL scatter(newlo_pp,utemp_pp) ; loads_pp = newlo_pp * globma_pp
    newlo_pp = .0_iwp
    IF(j/npri*npri==j .AND. numpe==it)                  &
             WRITE(11,'(2E12.4)')real_time,loads_pp(is)
  END DO timesteps
 IF(numpe==it) WRITE(11,*)"This analysis took  :",  elap_time( )-timest(1)
 CALL shutdown( )
END PROGRAM p125
```

New dynamic real arrays:

`globma_pp`	distributed global mass matrix
`globma_tmp`	temporary storage mass vector
`mass`	element lumped mass
`newlo_pp`	distributed new loads
`store_pm_pp`	distributed pm matrices

```
nels    nxe  nze  nip
125000   50   50   8

aa     bb     cc     kx    ky    kz
0.02  0.02  0.02  1.0   1.0   1.0

dtim    nsteps
0.0002   5000

npri   val0
500    100.0
```

Figure 12.23 Data for Program 12.5 example

```
This job ran on    16  processors

There are   132651 nodes    7651 restrained and  125000 equations
 Time after setup  is  : 0.209999999999126885
 Time after element integration is : 8.25999999999476131
    Time     Pressure
  0.1000E+00  0.8551E+02
  0.2000E+00  0.4599E+02
  0.3000E+00  0.2228E+02
.
.
.
  0.9000E+00  0.2595E+00
  0.1000E+01  0.1231E+00
 This analysis took  : 1006.36000000000058
```

Figure 12.24 Results from Program 12.5 example

Mesh	No of Processors	Analysis Time(secs)
50x50x50	16	1006
	32	516

Figure 12.25 Performance statistics: Program 12.5 (IBM SP2)

As is often the case in the parallel programs, an extra loop `elements_0` is used to organise geometry and distribution of the freedoms via steering vector `g`. Thereafter loop `elements_1` with its embedded `gauss_pts` carries over from serial to parallel versions, as does loop `elements_2`. In parallel, results at the appropriate freedom, `is`, are printed from processor `it`. Data are listed as Figure 12.23 with results as Figure 12.24 and performance statistics as Figure 12.25.

Program 12.6 Three dimensional steady state Navier–Stokes analysis. Compare Program 9.2.

```
PROGRAM p126
!-----------------------------------------------------------------------
!      Program 9.2 steady state  3-d Navier-Stokes equation
!      using 20-node velocity hexahedral elements  ; ns_cube
!      coupled to 8-node pressure hexahedral elements ; u-p-v-w order
!      element by element solution using BiCGSTAB(L) ; parallel version
!-----------------------------------------------------------------------
 USE new_library; USE geometry_lib; USE precision; USE utility
 USE mp_module  ; USE timing; USE global_variables1
 USE gather_scatter6;  IMPLICIT NONE
! ndof,nels,neq,ntot are now global variables - not declared
```

```
 INTEGER::nxe,nye,nze,nn,nip,nodof=4,nod=20,nodf=8,ndim=3, cj_tot,        &
          i,j,k,l,iel,ell,limit ,fixed_nodes,iters, cjiters , cjits,      &
          nr,n_t,num_no,no_index_start, nn_temp,neq_temp, nres , is,it
 REAL(iwp)::visc, rho, rho1,det,ubar, vbar, wbar,tol ,cjtol, alpha,beta,  &
       aa,bb,cc,penalty , x0 , pp , kappa,gama,omega,norm_r,r0_norm,error
 LOGICAL :: converged,cj_converged
 CHARACTER(LEN=15) :: element = 'hexahedron'
!---------------------------- dynamic arrays----------------------------
 REAL(iwp),ALLOCATABLE ::points(:,:), coord(:,:),derivf(:,:),fun(:),      &
                    jac(:,:),kay(:,:),der(:,:),deriv(:,:),weights(:),     &
                    derf(:,:),funf(:),coordf(:,:),p_g_co_pp(:,:,:),       &
                    c11(:,:),c21(:,:),c12(:,:),val(:),wvel(:),ke(:,:),    &
                    c23(:,:),c32(:,:),x_pp(:),b_pp(:),r_pp(:,:),          &
                    funny(:,:),row1(:,:),row2(:,:),uvel(:),vvel(:),       &
                    funnyf(:,:),rowf(:,:),storke_pp(:,:,:),diag_pp(:),    &
                    utemp_pp(:,:),xold_pp(:),c24(:,:),c42(:,:),row3(:,:),&
                    u_pp(:,:),rt_pp(:),y_pp(:),y1_pp(:),s(:),Gamma(:),    &
                    GG(:,:), diag_tmp(:,:),store_pp(:),pmul_pp(:,:)
 INTEGER, ALLOCATABLE :: rest(:,:),g(:),num(:),g_num_pp(:,:),g_g_pp(:,:) ,&
                        no(:),g_t(:),no_local(:), no_local_temp(:)
!---------------------------input and initialisation---------------------
  timest(1) = elap_time( )      ; cj_tot = 0
  CALL find_pe_procs(numpe,npes)
  IF(numpe==npes) THEN
    OPEN (10,FILE='p126.dat',STATUS=    'OLD',ACTION='READ')
    READ (10,*) nels,nxe,nze,nip,aa,bb,cc,                        &
             visc,rho,tol,limit , cjtol,cjits , penalty , x0, ell, kappa
  END IF
  CALL bcast_inputdata_p126(numpe,npes,nels,nxe,nze,nip,aa,bb,cc,         &
              visc,rho,tol,limit,cjtol,cjits,penalty,x0,ell,kappa)
   CALL calc_nels_pp; ntot=nod+nodf+nod+nod ; n_t=nod*nodof; neq_temp = 0
   nn_temp = 0 ; nye=nels/nxe/nze; fixed_nodes=3*nxe*nye+2*nxe+2*nye+1
   nr=3*nxe*nye*nze + 4*(nxe*nye+nye*nze+nze*nxe) + nxe+nye+nze + 2
  ALLOCATE (points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),fun(nod),  &
           jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),  &
           derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),funny(nod,1),    &
           g_g_pp(ntot,nels_pp),c11(nod,nod),c12(nod,nodf),c21(nodf,nod),&
           ke(ntot,ntot),rest(nr,nodof+1),c24(nodf,nod),c42(nod,nodf),   &
           p_g_co_pp(nod,ndim,nels_pp),g_num_pp(nod,nels_pp),num(nod),   &
           c32(nod,nodf),c23(nodf,nod),uvel(nod),vvel(nod),row1(1,nod),  &
           funnyf(nodf,1),rowf(1,nodf),no_local_temp(fixed_nodes),       &
           storke_pp(ntot,ntot,nels_pp),wvel(nod), row3(1,nod),g_t(n_t), &
           s(ell+1),GG(ell+1,ell+1),g(ntot), Gamma(ell+1),weights(nip),  &
           no(fixed_nodes),val(fixed_nodes),diag_tmp(ntot,nels_pp),      &
           utemp_pp(ntot,nels_pp),pmul_pp(ntot,nels_pp),row2(1,nod))
      uvel =.0_iwp; vvel =.0_iwp ; wvel = .0_iwp  ; ielpe = iel_start
      kay=0.0_iwp; kay(1,1)=visc/rho; kay(2,2)=visc/rho; kay(3,3)=visc/rho
      CALL ns_cube_bc20(nxe,nye,nze,rest); CALL ns_loading(nxe,nye,nze,no)
      CALL sample(element,points,weights) ; CALL rearrange(rest)
!----------------- loop the elements to set up global arrays--------------
 elements_1: DO iel = 1 , nels_pp
            CALL geometry_20bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
            CALL find_g3(num,g_t,rest) ; CALL g_t_g_ns(nod,g_t,g)
            p_g_co_pp(:,:,iel)=coord;g_g_pp(:,iel)=g; ielpe=ielpe+1
            i = MAXVAL(g); j = MAXVAL(num) ; g_num_pp(:,iel)=num
            IF(i>neq_temp) neq_temp = i; IF(j>nn_temp) nn_temp = j
 END DO elements_1
 neq = reduce(neq_temp); nn = reduce(nn_temp)
 CALL calc_neq_pp  ;  CALL make_ggl(g_g_pp) ; nres = nze*(nxe+1)+3*nxe+1
 DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN; it=numpe;is=i; END IF;END DO
 IF(numpe==it) THEN
    OPEN (11,FILE='p126.res',STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I5,A)') "This job ran on ",npes,"  processors"
    WRITE(11,'(A)') "Global coordinates and node numbers"
    DO i=1,nels_pp,nels_pp-1
                  WRITE(11,'(A,I8)') "Element  ",i; num = g_num_pp(:,i)
    DO k=1, nod;  WRITE(11,'(A,I8,3E12.4)')                               &
                "Node",num(k),p_g_co_pp(k,:,i); END DO
    END DO
  WRITE(11,'(A,3(I8,A))')"There are ",nn,"  nodes ",nr,                   &
                         "  restrained and  ",  neq, "  equations"
  WRITE(11,*) "Time after setup is   :" , elap_time( ) - timest(1)
 END IF
!---------------------- organise fixed nodes ---------------------------
  CALL reindex_fixed_nodes                                                &
             (ieq_start,no,no_local_temp,num_no,no_index_start)
  ALLOCATE(no_local(1:num_no)); no_local = no_local_temp(1:num_no)
  DEALLOCATE(no_local_temp)
  ALLOCATE(x_pp(neq_pp),rt_pp(neq_pp),r_pp(neq_pp,ell+1),                 &
           u_pp(neq_pp,ell+1),b_pp(neq_pp),diag_pp(neq_pp),               &
           xold_pp(neq_pp),y_pp(neq_pp),y1_pp(neq_pp),store_pp(neq_pp))
     iters = 0 ; x_pp=.0_iwp;  xold_pp = .0_iwp  ; val = 1.0_iwp
!------------------------main iteration loop  --------------------------
  iterations: DO     ;               iters = iters + 1
  ke=.0_iwp ; diag_pp=.0_iwp; b_pp=.0_iwp; utemp_pp=.0_iwp; pmul_pp=.0_iwp
  CALL gather(x_pp,utemp_pp); CALL gather(xold_pp,pmul_pp)
!------------ element stiffness integration and storage ------------------
      elements_2:  DO iel = 1 , nels_pp
              coord=p_g_co_pp(:,:,iel)
              coordf(1:4,:)=coord(1:7:2,:);coordf(5:8,:)=coord(13:19:2,:)
              uvel = (utemp_pp(1:nod,iel)+pmul_pp(1:nod,iel))*.5_iwp
              DO i = nod + nodf + 1 , nod + nodf + nod
                vvel(i-nod-nodf)=(utemp_pp(i,iel)+pmul_pp(i,iel))*.5_iwp
              END DO
              DO i = nod + nodf + nod + 1 , ntot
              wvel(i-nod-nodf-nod)=(utemp_pp(i,iel)+pmul_pp(i,iel))*.5_iwp
              END DO
              c11= .0_iwp; c12= .0_iwp; c21= .0_iwp; c23 = .0_iwp
              c32 = .0_iwp; c24=.0_iwp ; c42= .0_iwp
           gauss_points_1: DO i = 1 , nip
!--------------------- velocity contribution ---------------------------
              CALL shape_fun(fun,points,i) ;funny(:,1) = fun
              ubar = DOT_PRODUCT(fun,uvel);vbar = DOT_PRODUCT(fun,vvel)
              wbar = DOT_PRODUCT(fun,wvel)
              IF(iters==1)THEN;ubar=1._iwp;vbar=0._iwp;wbar=.0_iwp; END IF
              CALL shape_der(der,points,i);  jac = MATMUL(der,coord)
              det = determinant(jac )      ; CALL invert(jac)
              deriv = MATMUL(jac,der) ; row1(1,:) = deriv(1,:)
              row2(1,:)=deriv(2,:)     ; row3(1,:) = deriv(3,:)
              c11 = c11 + MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)      &
                     *det* weights(i) + &
                          MATMUL(funny,row1)*det*weights(i)*ubar +        &
                          MATMUL(funny,row2)*det*weights(i)*vbar +        &
                          MATMUL(funny,row3)*det*weights(i)*wbar
!---------------------now the pressure contribution---------------------
               CALL shape_fun(funf,points,i); funnyf(:,1)=funf
               CALL shape_der(derf,points,i)  ;jac=MATMUL(derf,coordf)
               det=determinant(jac)      ;     CALL invert(jac)
               derivf=MATMUL(jac,derf)   ;     rowf(1,:) = derivf(1,:)
               c12 = c12 + MATMUL(funny,rowf)*det*weights(i)/rho
               rowf(1,:) = derivf(2,:)
               c32 = c32 + MATMUL(funny,rowf)*det*weights(i)/rho
               rowf(1,:) = derivf(3,:)
               c42 = c42 + MATMUL(funny,rowf)*det*weights(i)/rho
               c21 = c21 + MATMUL(funnyf,row1)*det*weights(i)
               c23 = c23 + MATMUL(funnyf,row2)*det*weights(i)
               c24 = c24 + MATMUL(funnyf,row3)*det*weights(i)
           END DO gauss_points_1
      CALL formupvw(ke,c11,c12,c21,c23,c32,c24,c42); storke_pp(:,:,iel)=ke
      END DO elements_2   ; diag_tmp = .0_iwp
      elements_2a: DO iel = 1 , nels_pp
         DO k=1,ntot; diag_tmp(k,iel) = diag_tmp(k,iel) + ke(k,k) ; END DO
      END DO elements_2a    ;      CALL scatter(diag_pp,diag_tmp)
!----------- prescribed values of velocity and pressure ------------------
      DO i=1,num_no   ; k = no_local(i) - ieq_start +1
        diag_pp(k)=diag_pp(k)+penalty
        b_pp(k)=diag_pp(k)*val(no_index_start+i-1);store_pp(k)= diag_pp(k)
      END DO
!--- solve the simultaneous equations element by element using BiCGSTAB---
!-------------------      initialisation phase   ------------------------
    IF(iters==1) x_pp = x0  ;  pmul_pp = .0_iwp
    y_pp = x_pp  ;   y1_pp = .0_iwp ; CALL gather(y_pp,pmul_pp)
    elements_3 : DO iel = 1 , nels_pp
                  ke = storke_pp( : , : , iel)
                  utemp_pp(:,iel)= MATMUL(ke,pmul_pp(:,iel))
    END DO elements_3    ; CALL scatter(y1_pp,utemp_pp)
    DO i=1,num_no   ; k=no_local(i)-ieq_start+1
       y1_pp(k) = y_pp(k) * store_pp(k)
    END DO
    y_pp=y1_pp; rt_pp = b_pp - y_pp
    r_pp=.0_iwp;r_pp(:,1)=rt_pp ;u_pp=.0_iwp; gama=1.0_iwp; omega=1.0_iwp
    k = 0;norm_r=norm_p(rt_pp);r0_norm=norm_r; error=1.0_iwp;cjiters = 0
!------------------   bicgstab(ell)  iterations  ------------------------
        bicg_iterations : DO
            cjiters = cjiters + 1 ; cj_converged = error  < cjtol
            IF(cjiters==cjits.OR. cj_converged) EXIT
            gama = - omega*gama  ;  y_pp = r_pp(:,1)
            DO j = 1 , ell
               rho1 = DOT_PRODUCT_P(rt_pp,y_pp)  ;  beta = rho1/gama
               u_pp(:,1:j)=r_pp(:,1:j)-beta * u_pp(:,1:j); pmul_pp=.0_iwp
               y_pp = u_pp(:,j); y1_pp = .0_iwp; CALL gather(y_pp,pmul_pp)
               elements_4 : DO iel = 1 , nels_pp
                  ke = storke_pp(: , : , iel)
                  utemp_pp(:,iel)=MATMUL(ke,pmul_pp(:,iel))
               END DO elements_4  ;  CALL scatter(y1_pp,utemp_pp)
               DO i=1,num_no  ; l=no_local(i) -ieq_start +1
                   y1_pp(l) = y_pp(l) * store_pp(l)
               END DO
               y_pp=y1_pp; u_pp(:,j+1) = y_pp
               gama = DOT_PRODUCT_P(rt_pp,y_pp) ; alpha = rho1/gama
               x_pp=x_pp+ alpha * u_pp(:,1)
               r_pp(:,1:j)= r_pp(:,1:j)-alpha*u_pp(:,2:j+1);pmul_pp=.0_iwp
               y_pp = r_pp(:,j); y1_pp = .0_iwp; CALL gather(y_pp,pmul_pp)
               elements_5 : DO iel = 1 , nels_pp
```

```
                 ke=storke_pp(:,:, iel )
                 utemp_pp(:,iel) = MATMUL(ke,pmul_pp(:,iel))
               END DO elements_5 ; CALL scatter(y1_pp,utemp_pp)
               DO i=1,num_no  ; l = no_local(i) - ieq_start + 1
                     y1_pp(l) = y_pp(l) * store_pp(l)
               END DO
               y_pp=y1_pp ; r_pp(:,j+1) = y_pp
            END DO
            DO i=1,ell+1 ; DO j=1,ell+1
               GG(i,j) = DOT_PRODUCT_P(r_pp(:,i),r_pp(:,j))
            END DO ; END DO
            CALL form_s(GG,ell,kappa,omega,Gamma,s)
            x_pp = x_pp - MATMUL(r_pp,s);r_pp(:,1)=MATMUL(r_pp,Gamma)
            u_pp(:,1)=MATMUL(u_pp,Gamma)
            norm_r = norm_p(r_pp(:,1));  error = norm_r/r0_norm ;k= k + 1
        END DO bicg_iterations
!----------------------- end of BiCGSTAB(L) process-----------------------
    b_pp = x_pp - xold_pp ; pp = norm_p(b_pp) ; cj_tot = cj_tot +cjiters
   IF(numpe==it) THEN
    WRITE(11,'(A,E12.4)') "Norm of the error is :", pp
    WRITE(11,'(A,I5,A)')"It took BiCGSTAB(L) ",                       &
                         cjiters," iterations to converge"
   END IF
   CALL checon_par(x_pp,xold_pp,tol,converged,neq_pp)
   IF(converged.OR.iters==limit) EXIT
  END DO iterations
 IF(numpe==it) THEN
 WRITE(11,'(A)') " The pressure at the corner of the box is   :"
 WRITE(11,'(A)')" Freedom   Pressure   "
         WRITE(11,'(I8,E12.4)') nres, x_pp(is)
 WRITE(11,'(A,I5)') "The total number of BiCGStab iterations was :",cj_tot
 WRITE(11,'(A,I5,A)')"The solution took",iters,"  iterations to converge"
 END IF
 IF(numpe==it) WRITE(11,*) "This analysis took  :" ,elap_time( )-timest(1)
 CALL shutdown( )
end program p126
```

New scalar integers:

`cj_tot`	total number of BiCGStab iterations
`ell`	`l` in the BiCGStab(l) process
`fixed_nodes`	number of fixed nodes
`l`	simple counter
`nodf`	number of pressure degrees of freedom per element
`n_t`	total number of degrees of freedom per element

New scalar reals:

`error`	residual error
`gama`	intermediate value
`kappa`	`kappa` in BiCGStab process
`norm_r`	norm of residual
`omega`	intermediate value
`pp`	intermediate value
`rho`	density
`rho1`	intermediate value
`r0_norm`	starting residual norm

`ubar`	average x velocity
`vbar`	average y velocity
`visc`	viscosity
`wbar`	average z velocity
`x0`	start value

New dynamic integer arrays:

`g_t`	total element g vector

New dynamic real arrays:

`b_pp`	distributed right-hand side vector
`coordf`	nodal coordinates of pressure nodes
`c11`	
`c12`	
`c21`	
`c23`	c arrays—see equation (2.115)
`c32`	
`c24`	
`c42`	
`derf`	local derivatives of pressure shape functions
`derivf`	global derivatives of pressure shape functions
`diag_pp`	distributed diagonal vector
`funf`	intermediate array
`funnyf`	intermediate array
`gamma`	intermediate array
`gg`	intermediate array
`ke`	element "stiffness" matrix
`rowf`	row of fluid derivative matrix
`row1`	
`row2`	intermediate arrays
`row3`	
`rt_pp`	distributed vector
`s`	intermediate array
`storke_pp`	distributed `ke` matrices
`uvel`	x-velocity
`vvel`	y-velocity
`wvel`	z-velocity
`xold_pp`	distributed previous x vector
`y_pp`	distributed y vector
`y1_pp`	distributed y1 vector

In parallel, boundary restraint data are created by `ns_cube_bc20` and loading by `ns_loading` for the special case of the cuboidal lid-driven cavity problem. Loops `elements_1` and `elements_2`, with embedded `gauss_pts_1` carry over from serial to parallel, the differences being due to the extension from 2D (serial) to 3D (parallel). For example `formupv` (serial) becomes `formupvw` (parallel). After initialisation involving loop `elements_3`, the BiCGStab iterations with loops `elements_4` and `elements_5`

```
nels    nxe  nze  nip
8000    20   20    8

aa     bb    cc
0.5    0.5   0.5

visc  rho   tol   limit
0.01  1.0  0.001   30

cjtol      cjits   penalty
1.0e-5      500     1.0e5

x0   ell   kappa
1.0   4     0.0
```

Figure 12.26 Data for Program 12.6 example

```
This job ran on    16  processors

There are    35721  nodes    28862  restrained and     96078  equations
 Time after setup is   : 0.900000000037834980E-01
Norm of the error is :  0.1533E+03
It took BiCGSTAB(L)   165 iterations to converge
Norm of the error is :  0.3581E+02
It took BiCGSTAB(L)   151 iterations to converge
Norm of the error is :  0.8611E+01
It took BiCGSTAB(L)   146 iterations to converge
.
.
.
Norm of the error is :  0.1296E+00
It took BiCGSTAB(L)   155 iterations to converge
Norm of the error is :  0.7125E-01
It took BiCGSTAB(L)   149 iterations to converge
 The pressure at the corner of the box is   :
 Freedom   Pressure
     481  0.1110E+01
The total number of BiCGStab iterations was : 1384
The solution took    9  iterations to converge
 This analysis took  : 1153.42000000000553
```

Figure 12.27 Results from Program 12.6 example

Elements	Iterations to convergence
512	672
1,000	713
1,728	900
8,000	1,385
64,000	2,508
125,000	3,313

Figure 12.28 Problem size vs. Number of iterations: Program 12.6 (SGI Origin 3000)

can be traced clearly between the two versions. In parallel, only the pressure at the corner of the box is printed. Data are listed as Figure 12.26 with results as Figure 12.27. The effect of problem size on the number of iterations to convergence is shown in Figure 12.28 and some performance statistics are listed in Figure 12.29. Speed-up versus number of processors for larger data sets are shown in Figure 12.30. Post processing of results is now critical and a typical display is shown as Figure 12.31.

```
Mesh          No of Processors   Analysis Time(secs)

20x20x20            16                  1153
                    32                   596
```

Figure 12.29 Performance statistics: Program 12.6 (IBM SP2)

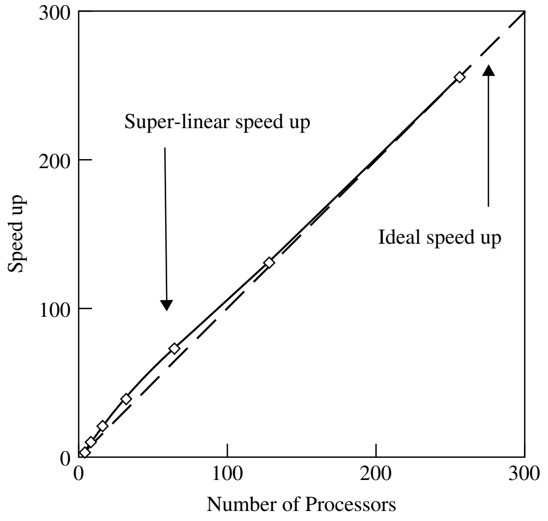


Figure 12.30 Speedup versus number of Processors with 4.5 million equations: Program 12.6 (SGI Origin 3000)

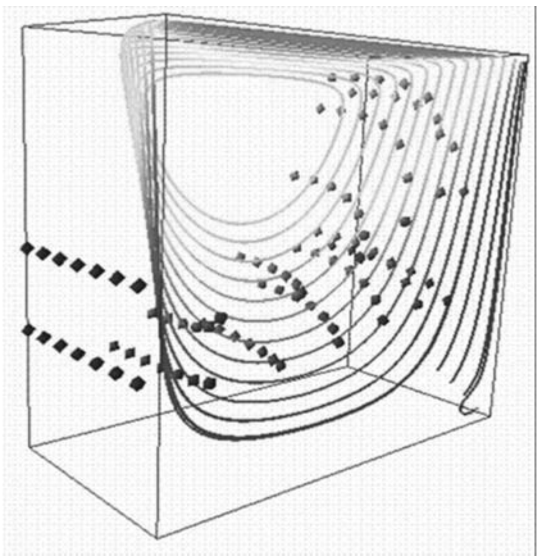


Figure 12.31 Typical displays (Margetts 2002): Program 12.6 (*Continued on page 551*)

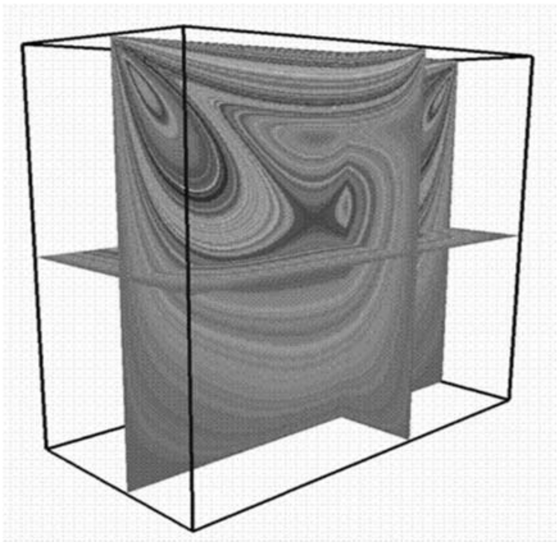


Figure 12.31 (*Continued from page 550*)

Program 12.7 Three-dimensional analysis of Biot poro-elastic solid. Compare Program 9.2.

```
PROGRAM p127
!-------------------------------------------------------------------
!      Program 9.5 3 - D  consolidation of a cuboidal Biot elastic
!      solid using 20 -node solid hexahedral elements  coupled to 8-node
!      fluid elements  -  parallel pcg version   - biot_cube
!-------------------------------------------------------------------
 USE precision; USE mp_module;USE gather_scatter6; USE utility; USE timing
 USE global_variables1; USE new_library; USE geometry_lib ; IMPLICIT NONE
 ! nels,neq,ntot,ndof are now in global_variables1
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=4,nod=20,nodf=8,nst=6,ndim=3,       &
          i,j,k,l,iel, ns,nstep,cjiters,cjits,loaded_freedoms,            &
          num_no,no_index_start,n_t, neq_temp, nn_temp, nle
 REAL(iwp)::kx,ky,kz,e,v,det,dtim,theta,real_time,                        &
            up,alpha,beta,cjtol,aa,bb,cc ,q
 LOGICAL :: cj_converged   ;     CHARACTER(LEN=15) :: element='hexahedron'
!---------------------------- dynamic arrays-----------------------------
 REAL(iwp) ,ALLOCATABLE :: dee(:,:),points(:,:),coord(:,:),derivf(:,:),   &
                       jac(:,:),kay(:,:),der(:,:),deriv(:,:),weights(:),  &
                       derf(:,:),funf(:), coordf(:,:), bee(:,:), km(:,:), &
                       eld(:), sigma(:), kc(:,:),ke(:,:),p_g_co_pp(:,:,:),&
                       kd(:,:),fun(:),c(:,:),loads_pp(:),pmul_pp(:,:),    &
                       vol(:), storke_pp(:,:,:), ans_pp(:) ,volf(:,:) ,   &
                       p_pp(:),x_pp(:),xnew_pp(:),u_pp(:),eld_pp(:,:),    &
                       diag_precon_pp(:),diag_precon_tmp(:,:),d_pp(:),    &
                       utemp_pp(:,:), storkd_pp(:,:,:),val(:)
 INTEGER, ALLOCATABLE :: rest(:,:),g(:),num(:),g_g_pp(:,:),g_num_pp(:,:), &
                         g_t(:),no(:),no_local_temp(:),no_local(:)
!------------------------input and initialisation-----------------------
 timest(1) = elap_time( )     ;   CALL find_pe_procs(numpe,npes)
 IF(numpe==npes)THEN
```

```
  OPEN (10,FILE='p127.dat',STATUS=    'OLD',ACTION='READ')
  READ (10,*) nels,nxe,nze,aa,bb,cc,nip, kx, ky, kz, e,v,     &
              dtim, nstep, theta , cjits , cjtol
 END IF
 CALL bcast_inputdata_p127(numpe,npes,nels,nxe,nze,aa,bb,cc,nip,kx,   &
             ky,kz,e,v,dtim,nstep,theta,cjits,cjtol)
 CALL calc_nels_pp;ndof=nod*ndim; ntot=ndof+nodf; neq_temp= 0; nn_temp= 0
 n_t=nod*nodof ; nye = nels/nxe/nze    ; nle = nxe/5
 nr = 3*nxe*nye*nze + 4*(nxe*nye+nye*nze+nze*nxe) + nxe+nye+nze + 2
 loaded_freedoms = 3*nle*nle + 4*nle + 1
 ALLOCATE(dee(nst,nst),points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),&
        jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),      &
        derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),bee(nst,ndof),       &
        km(ndof,ndof),eld(ndof),sigma(nst),kc(nodf,nodf),weights(nip),    &
        g_g_pp(ntot,nels_pp),diag_precon_tmp(ntot,nels_pp),               &
        ke(ntot,ntot),kd(ntot,ntot),fun(nod),c(ndof,nodf),g_t(n_t),       &
        vol(ndof),rest(nr,nodof+1), g(ntot), volf(ndof,nodf),             &
        p_g_co_pp(nod,ndim,nels_pp),g_num_pp(nod,nels_pp),num(nod),       &
        storke_pp(ntot,ntot,nels_pp),storkd_pp(ntot,ntot,nels_pp),        &
        pmul_pp(ntot,nels_pp),utemp_pp(ntot,nels_pp),eld_pp(ntot,nels_pp),&
        no(loaded_freedoms),val(loaded_freedoms),                         &
        no_local_temp(loaded_freedoms))
           kay=0.0_iwp; kay(1,1)=kx; kay(2,2)=ky ; kay(3,3) = kz
  CALL biot_cube_bc20(nxe,nye,nze,rest)  ; CALL rearrange(rest)
  CALL biot_loading(nxe,nze,nle,no,val)  ;  val = -val * aa * bb / 12._iwp
  CALL sample(element,points,weights);CALL deemat(dee,e,v);ielpe=iel_start
!----------------- loop the elements to  set up global arrays-------------
 elements_1: DO iel = 1 , nels_pp
             CALL geometry_20bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
             CALL find_g3(num,g_t,rest); CALL g_t_g(nod,g_t,g)
             p_g_co_pp(:,:,iel)=coord; g_g_pp(:,iel)=g; ielpe = ielpe + 1
             i = MAXVAL(g); j = MAXVAL(num); g_num_pp(:,iel) = num
             IF(i>neq_temp)neq_temp = i; IF(j>nn_temp)nn_temp = j
 END DO elements_1
 neq = reduce(neq_temp); nn = reduce(nn_temp)
 CALL calc_neq_pp;  CALL make_ggl(g_g_pp)
 IF(numpe==1) THEN
   OPEN (11,FILE='p127.res',STATUS='REPLACE',ACTION='WRITE')
   WRITE(11,'(A,I5,A)') "This job ran on ", npes, "  processors"
   WRITE(11,'(A)') "Global coordinates and node numbers"
   DO i=1,nels_pp,nels_pp-1; WRITE(11,'(A,I8)')"Element ",i
    num=g_num_pp(:,i)
    DO k=1,nod;WRITE(11,'(A,I8,3E12.4)')                                  &
                      "   Node",num(k),p_g_co_pp(k,:,i);END DO
   END DO
   WRITE(11,'(A,3(I8,A))') "There are ",nn, " nodes", nr,                &
                         " restrained and ",&
    neq, "  equations "
   WRITE(11,*) "Time after setup is   :",  elap_time( ) - timest(1)
 END IF
 ALLOCATE(loads_pp(neq_pp),ans_pp(neq_pp),p_pp(neq_pp),x_pp(neq_pp),      &
         xnew_pp(neq_pp),u_pp(neq_pp),diag_precon_pp(neq_pp),d_pp(neq_pp))
     loads_pp = .0_iwp ; p_pp = .0_iwp; xnew_pp = .0_iwp
     diag_precon_pp = .0_iwp ;  diag_precon_tmp =.0_iwp
!-------- element stiffness integration , storage and preconditioner -----
      elements_2:  DO iel = 1 , nels_pp
              coord = p_g_co_pp(:,:,iel)
              coordf(1:4,:)=coord(1:7:2,:);coordf(5:8,:)=coord(13:20:2,:)
```

```
               km = .0_iwp; c = .0_iwp; kc = .0_iwp
           gauss_points_1: DO i = 1 , nip
              CALL shape_der(der,points,i);  jac = MATMUL(der,coord)
              det=determinant(jac);CALL invert(jac);deriv=MATMUL(jac,der)
              CALL beemat(bee,deriv); vol(:)=bee(1,:)+bee(2,:)+bee(3,:)
              km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det* weights(i)
!--------------------------now the fluid contribution---------------------
               CALL shape_fun(funf,points,i)
               CALL shape_der(derf,points,i)  ; derivf=MATMUL(jac,derf)
    kc=kc+MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtim
               DO l=1,nodf; volf(:,l)=vol(:)*funf(l); END DO
               c= c+volf*det*weights(i)
           END DO gauss_points_1  ;     CALL fmkdke(km,kc,c,ke,kd,theta)
          storke_pp(: , : , iel) = ke   ;    storkd_pp( : , : , iel ) = kd
          DO k=1,ndof
           diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)+theta*km(k,k)
          END DO
          DO k=1 , nodf
           diag_precon_tmp(ndof+k,iel)=diag_precon_tmp(ndof+k,iel)        &
                                      -theta*theta*kc(k,k)
          END DO
      END DO elements_2   ;   CALL scatter(diag_precon_pp,diag_precon_tmp)
          diag_precon_pp=1._iwp/diag_precon_pp;DEALLOCATE(diag_precon_tmp)
!----------------------------loaded freedoms -----------------------------
      CALL reindex_fixed_nodes                                            &
                 (ieq_start,no,no_local_temp,num_no,no_index_start)
      ALLOCATE (no_local(1:num_no)) ; no_local = no_local_temp(1:num_no)
      DEALLOCATE(no_local_temp)
! ------------------------ enter the time-stepping loop-------------------
      real_time = .0_iwp
  time_steps:  DO ns = 1 , nstep
      ans_pp = .0_iwp    ;   real_time=real_time+dtim
    IF(numpe==1)THEN;WRITE(11,'(A,E12.4)')"The time is",real_time ; END IF
    pmul_pp=.0_iwp ; utemp_pp=.0_iwp ;  CALL gather(loads_pp,pmul_pp)
      elements_3: DO iel = 1 , nels_pp
        utemp_pp(:,iel)=MATMUL(storkd_pp(:,:,iel),pmul_pp(:,iel))
      END DO elements_3       ;    CALL scatter(ans_pp,utemp_pp)
!-------------------------   ramp loading  -------------------------------
    IF(ns>10) THEN
      DO i=1,num_no ; j = no_local(i)-ieq_start + 1
         ans_pp(j)=ans_pp(j) + val(no_index_start+i-1)
      END DO
    ELSE IF (ns<=10) THEN
      DO i=1,num_no    ;  j = no_local(i)-ieq_start+1
         ans_pp(j)=ans_pp(j)+val(no_index_start+i-1)*(.1_iwp*ns+.1_iwp*   &
                                 (theta-1._iwp))
      END DO
    END IF
      d_pp = diag_precon_pp*ans_pp; p_pp = d_pp
      x_pp = .0_iwp  ! depends on starting x = .0
!-----------------   solve the simultaneous equations by pcg -------------
       cjiters = 0
      conjugate_gradients:  DO
       cjiters = cjiters + 1 ; u_pp = .0_iwp
    pmul_pp=.0_iwp ; u_pp=.0_iwp ;    CALL gather(p_pp,pmul_pp)
      elements_4 : DO iel = 1 , nels_pp
        utemp_pp(:,iel)=MATMUL(storke_pp(:,:,iel),pmul_pp(:,iel))
      END DO elements_4        ;    CALL scatter(u_pp,utemp_pp)
```

```
!----------------------------pcg process --------------------------------
    up =DOT_PRODUCT_P(ans_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
    xnew_pp = x_pp + p_pp* alpha; ans_pp = ans_pp - u_pp*alpha
    d_pp = diag_precon_pp*ans_pp
    beta = DOT_PRODUCT_P(ans_pp,d_pp)/up; p_pp = d_pp + p_pp * beta
    CALL checon_par(xnew_pp,x_pp,cjtol,cj_converged,neq_pp)
    IF(cj_converged.OR.cjiters==cjits) EXIT
 END DO conjugate_gradients
!----------- end of pcg process-------------------------------------------
  ans_pp = xnew_pp; loads_pp = ans_pp
  IF(numpe==1) THEN
     WRITE(11,'(A,I5,A)')                                                &
          "Conjugate gradients took ",cjiters, "  iterations to converge"
     WRITE(11,'(A)') " The nodal displacements and porepressures are    :"
     WRITE(11,'(4E12.4)') ans_pp(1:4)
  END IF
!-------------------recover stresses at  gauss-points --------------------
          eld_pp = .0_iwp;  CALL gather(ans_pp,eld_pp)  ; iel = 1
          coord=p_g_co_pp(:,:,iel)  ; eld = eld_pp(:,iel)
            IF(numpe==1) WRITE(11,'(A,I5,A)')                            &
              "The Gauss Point effective stresses for element",iel,"  are"
            gauss_pts_2: DO i = 1,nip
              CALL shape_der (der,points,i);  jac= MATMUL(der,coord)
              CALL invert ( jac );    deriv= MATMUL(jac,der)
              CALL beemat(bee,deriv);sigma= MATMUL(dee,MATMUL(bee,eld))
              IF(numpe==1.AND.i==1) THEN
               WRITE(11,'(A,I5)') "Point  ",i;WRITE(11,'(6E12.4)') sigma
              END IF
            END DO gauss_pts_2
    END DO time_steps
  IF(numpe==1) WRITE(11,*) "This analysis took  :", elap_time( )-timest(1)
  CALL shutdown( )
END PROGRAM p127
```

New scalar integer:

`ns`	timestep counter

New dynamic real arrays:

`ans_pp`	distributed answer vector
`c`	coupling matrix
`kd`	total element matrix
`storkd_pp`	distributed `kd` matrices
`vol`	array for volumetric strain
`volf`	array for fluid volumetric strain

Again the main differences between serial and parallel programs relate to the change in geometry from 2D to 3D. Loop `elements_1` uses `geometry_20bxz` in place of `geom_rect` in serial and loop `elements_2` with embedded `gauss_pts_1` is almost identical although the parallel version assumes constant element properties. Ramp loading is also assumed rather than the general pattern allowed in serial (2D). Loop `elements_4` carries over from serial to parallel but in the latter case only a few surface displacements are printed and only the stresses in the “first” (central surface) element are computed and printed. Data are listed as Figure 12.32 with results as Figure 12.33 and performance statistics as Figure 12.34.

```
nels    nxe  nze
8000    20   20

aa     bb    cc       nip
0.5    0.5   0.5       8

kx   ky   kz
1.0  1.0  1.0

e     v
1.0  0.0

dtim  nstep  theta
1.0    20     1.0

cjits  cjtol
1000  0.00001
```

Figure 12.32 Data for Program 12.7 example

```
This job ran on 32  processors

There are    35721 nodes   28862 restrained and   107180  equations
 Time after setup is   : 0.230000000003201421
The time is  0.1000E+01
Conjugate gradients  took   360  iterations to converge
 The nodal displacements and porepressures are    :
 -0.2591E+00 -0.5907E-02 -0.2582E+00 -0.1201E-01
The Gauss Point effective stresses for element    1  are
Point      1
 -0.2061E-01 -0.2061E-01 -0.9442E-01  0.6251E-03 -0.2071E-03 -0.2071E-03
The time is  0.2000E+01
Conjugate gradients took   262  iterations to converge
 The nodal displacements and porepressures are    :
 -0.5495E+00 -0.1402E-01 -0.5478E+00 -0.2875E-01
The Gauss Point effective stresses for element    1  are
Point      1
 -0.5057E-01 -0.5057E-01 -0.1929E+00  0.3545E-02 -0.1009E-02 -0.1009E-02
.
.
.
The time is  0.2000E+02
Conjugate gradients took   313  iterations to converge
 The nodal displacements and porepressures are    :
 -0.3638E+01 -0.1177E+00 -0.3623E+01 -0.2354E+00
The Gauss Point effective stresses for element    1  are
Point      1
 -0.4145E+00 -0.4145E+00 -0.9953E+00  0.5245E-02  0.6459E-02  0.6459E-02
 This analysis took  : 255.710000000006403
```

Figure 12.33 Results from Program 12.7 example

Mesh	No of Processors	Analysis Time(secs)
20x20x20	16	538
	32	256

Figure 12.34 Performance statistics: Program 12.7 (IBM SP2)

Program 12.8 Eigenvalue analysis of three-dimensional elastic solid. Compare Program 10.4.

```
PROGRAM p128
!-------------------------------------------------------------------------
!      Program 10.4 eigenvalues and eigenvectors of a cuboidal elastic
!      solid in 3d using  uniform 8-node hexahedral elements
!      for lumped mass this is done element by element : parallel version
!-------------------------------------------------------------------------
 USE new_library ; USE geometry_lib ; USE lancz_lib; USE precision
 USE timing      ; USE utility; USE mp_module; USE global_variables1
 USE gather_scatter6   ; IMPLICIT NONE
! ndof,nels,neq,ntot are global - not declared
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=3,nod=8,nst=6,i,j,k,iel,ndim=3,     &
          nmodes,jflag,iflag=-1,lp=11,lalfa,leig,lx,lz,iters,neig=0
 REAL(iwp)::aa,bb,cc,rho,e,v,det  , el,er,  acc
 CHARACTER (LEN=15) :: element = 'hexahedron'
!-------------------------- dynamic arrays-------------------------------
 REAL(iwp),ALLOCATABLE:: points(:,:),dee(:,:),coord(:,:),vdiag_pp(:),     &
                        fun(:),jac(:,:),der(:,:),deriv(:,:),weights(:),  &
                        bee(:,:),km(:,:),emm(:,:),ecm(:,:),utemp_pp(:,:),&
                        ua_pp(:),va_pp(:),eig(:),del(:), udiag_pp(:),    &
                        diag_pp(:),alfa(:),beta(:),w1_pp(:),y_pp(:,:),   &
                        z_pp(:,:), pmul_pp(:,:),v_store_pp(:,:),         &
                        p_g_co_pp(:,:,:), diag_tmp(:,:),x(:)
 INTEGER, ALLOCATABLE  :: rest(:,:), g(:), num(:) , g_num_pp(:,:) ,       &
                         g_g_pp (:,:), nu(:),jeig(:,:)
!---------------------input and initialisation--------------------------
 timest(1) = elap_time( )  ; CALL find_pe_procs(numpe,npes)
 IF(numpe==npes) THEN
  OPEN (10,FILE='p128.dat',STATUS=    'OLD',ACTION='READ')
  OPEN (11,FILE='p128.res',STATUS='REPLACE',ACTION='WRITE')
  READ (10,*) nels,nxe,nze,nip,aa,bb,cc,rho,e,v,                          &
                    nmodes,el,er,lalfa,leig,lx,lz,acc
 END IF
 CALL bcast_inputdata_p128(numpe,npes,nels,nxe,nze,nip,aa,bb,cc,rho,e,v,  &
                            nmodes, el,er,lalfa,leig,lx,lz,acc)
 CALL calc_nels_pp ;  ndof=nod*nodof  ; ntot = ndof ; nn = 0; neq = 0
 nr = (nxe + 1) * (nze + 1)  ; nye = nels/nxe/nze
 ALLOCATE (rest(nr,nodof+1),points(nip,ndim),pmul_pp(ntot,nels_pp),      &
         coord(nod,ndim),fun(nod),jac(ndim,ndim), weights(nip),           &
         g_num_pp(nod,nels_pp),der(ndim,nod),deriv(ndim,nod),dee(nst,nst),&
         num(nod),km(ntot,ntot),g(ntot),g_g_pp(ntot,nels_pp),             &
         ecm(ntot,ntot),eig(leig),x(lx),del(lx),nu(lx),jeig(2,leig),      &
         alfa(lalfa),beta(lalfa),z_pp(lz,leig),utemp_pp(ntot,nels_pp),    &
         p_g_co_pp(nod,ndim,nels_pp),bee(nst,ntot),emm(ntot,ntot),        &
         diag_tmp(ntot,nels_pp))
   rest = 0 ;  DO i=1,nr; rest(i,1) = i; END DO  ;ielpe = iel_start
!------------------ loop the elements to set up global arrays  -----------
 elements_1 : DO iel =1,nels_pp
              CALL geometry_8bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
              CALL find_g(num,g,rest) ; g_num_pp(:,iel)=num
              p_g_co_pp(:,:,iel)=coord; g_g_pp(:,iel)=g ;ielpe=ielpe + 1
 END DO elements_1
 nn = (nxe+1)*(nze+1)*(nye+1)  ;  neq = (nn - nr)*nodof
 CALL calc_neq_pp   ;  CALL make_ggl(g_g_pp)
 IF(numpe==npes) THEN
    WRITE(11,'(A,I5,A)') "This job ran on ",npes, "   processors"
    WRITE(11,'(A)') "Global coordinates and node numbers"
    DO i=1,nels_pp,nels_pp-1; WRITE(11,'(A,I8)')"Element ",i
      num = g_num_pp(:,i)
      DO k=1,nod;WRITE(11,'(A,I8,3E12.4)')                              &
          "  Node",num(k),p_g_co_pp(k,:,i); END DO
    END DO
    WRITE(11,'(A,3(I8,A))') "There are ",nn," nodes",nr,               &
                            " restrained  and",  neq," equations"
    WRITE(11,*) "Time after setup is  :", elap_time( ) - timest(1)
 END IF
   ALLOCATE  ( ua_pp(neq_pp),va_pp(neq_pp),vdiag_pp(neq_pp),            &
              v_store_pp(neq_pp,lalfa),diag_pp(neq_pp),udiag_pp(neq_pp), &
              w1_pp(neq_pp), y_pp(neq_pp,leig))
    ua_pp = .0_iwp ; va_pp = .0_iwp  ; eig = .0_iwp  ; diag_tmp = .0_iwp
    jeig = 0;  x=.0_iwp; del=.0_iwp; nu=0; alfa=.0_iwp; beta=.0_iwp
    diag_pp=.0_iwp;udiag_pp=.0_iwp; w1_pp=.0_iwp; y_pp=.0_iwp; z_pp=.0_iwp
    CALL sample( element, points, weights); CALL deemat(dee,e,v)
!--------------- element stiffness integration and assembly---------------
 elements_2:DO iel=1,nels_pp
            coord = p_g_co_pp(:,:,iel); g = g_g_pp(: ,iel )
            km=0.0_iwp ; emm=0.0_iwp
           integrating_pts_1:  DO i=1,nip
            CALL shape_fun(fun,points,i)
            CALL shape_der(der,points,i); jac=MATMUL(der,coord)
            det= determinant(jac) ; CALL invert(jac)
            deriv = MATMUL(jac,der);CALL beemat(bee,deriv)
            km= km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
            CALL ecmat(ecm,fun,ntot,nodof);emm=emm+ecm*det*weights(i)*rho
           END DO integrating_pts_1
            DO k=1,ntot
              diag_tmp(k,iel)=diag_tmp(k,iel)+sum(emm(k,:));END DO
  END DO elements_2
  CALL scatter(diag_pp,diag_tmp); DEALLOCATE(diag_tmp)
!-----------------------------find eigenvalues---------------------------
  diag_pp = 1._iwp / sqrt(diag_pp) ! diag_pp holds l**(-1/2)
    DO iters = 1 , lalfa
     CALL lancz1(neq_pp,el,er,acc,leig,lx,lalfa,lp,iflag,ua_pp,va_pp,     &
                  eig,jeig,neig,x,del,nu,alfa,beta,v_store_pp)
       IF(iflag==0) EXIT
       IF(iflag>1) THEN
         IF(numpe==npes) THEN
          WRITE(11,'(A,I5)')                                               &
               " Lancz1 is signalling failure, with iflag = ",iflag;EXIT
         END IF
       END IF
!---- iflag = 1 therefore form u + a * v  ( done element by element )-----
      vdiag_pp= va_pp; vdiag_pp = vdiag_pp * diag_pp!vdiag is l**(-1/2).va
      udiag_pp = .0_iwp ; pmul_pp = .0_iwp; CALL gather(vdiag_pp,pmul_pp)
      elements_3 : DO iel = 1 , nels_pp
!                        utemp_pp(:,iel) = MATMUL(km,pmul_pp(:,iel))
                         CALL dgemv('n',ntot,ntot,1.0,                   &
                             km,ntot,pmul_pp(:,iel),1,0.0,utemp_pp(:,iel),1)
      END DO elements_3
      CALL scatter(udiag_pp,utemp_pp) !udiag is A.1**(-1/2).va
      udiag_pp = udiag_pp *  diag_pp  ; ua_pp = ua_pp + udiag_pp
    END DO
!-------------- iflag = 0 therefore write out the spectrum ---------------
    IF(numpe==npes) THEN
      WRITE(11,'(2(A,E12.4))') "The range is",el,"  to ",er
      WRITE(11,'(A,I8,A)') "There are ",neig," eigenvalues in the range"
      WRITE(11,'(A,I8,A)') "It took ",iters,"  iterations"
      WRITE(11,'(A)') "The eigenvalues are   :"
      WRITE(11,'(6E12.4)') eig(1:neig)
    END IF
!  calculate the eigenvectors
   IF(neig>10)neig = 10
   CALL lancz2(neq_pp,lalfa,lp,eig,jeig,neig,alfa,beta,lz,jflag,y_pp,     &
               w1_pp,z_pp,v_store_pp)
!------------------if jflag is zero  calculate the eigenvectors ----------
   IF (jflag==0) THEN
    IF(numpe==npes) THEN
     WRITE(11,'(A)') "The eigenvectors are  :"
     DO i = 1 , nmodes
        udiag_pp(:) = y_pp(:,i)  ; udiag_pp = udiag_pp * diag_pp
        WRITE(11,'("Eigenvector number  ",I4," is: ")')  i
        WRITE(11,'(6E12.4)') udiag_pp(1:6)
     END DO
   ELSE
! lancz2 fails
     WRITE(11,'(A,I5)')" Lancz2 is signalling failure with jflag = ",jflag
    END IF
   END IF
 IF(numpe==npes) WRITE(11,*)"This analysis took  :", elap_time()-timest(1)
 CALL shutdown( )
END PROGRAM p128
```

New scalar integers:

`iflag`	failure flag
`jflag`	failure flag
`lalfa`	length of `alfa` array
`leig`	length of `eig` array
`lp`	output channel number
`lx`	problem dependent array size
`lz`	problem dependent array size
`neig`	problem dependent array size
`nmodes`	number of eigenmodes computed

New scalar reals:

`acc`	accuracy parameter
`el`	left limit of eigenvalue spectrum
`er`	right limit of eigenvalue spectrum

`lancz1` and `lancz2` are aliases for `ep25a/ad` and `ep25e/ed` of HSL (2002).

New dynamic integer arrays:

`jeig`	intermediate array
`nu`	intermediate array

New dynamic real arrays:

`alfa`	intermediate array
`beta`	intermediate array
`del`	intermediate array
`diag_tmp`	diagonal mass matrix
`ecm`	element consistent mass matrix
`eig`	intermediate array
`emm`	accumulated element mass matrix
`ua_pp`	distributed $\{\mathbf{U}\}$ in product $\{\mathbf{U}\} + [\mathbf{A}]\{\mathbf{V}\}$
`va_pp`	distributed $\{\mathbf{V}\}$ in product $\{\mathbf{U}\} + [\mathbf{A}]\{\mathbf{V}\}$
`vdiag_pp`	distributed diagonal vector
`v_store_pp`	distributed stored Lanczos vectors
`w1_pp`	intermediate array
`x`	local element array
`z_pp`	intermediate array

Loop `elements_1` is a good illustration of the transition from serial to parallel code. Routine `geometry_8bxz`, to create a cuboidal mesh of 8-node hexahedra, replaces `geom_rect`, and `find_g` replaces `num_to_g`. Distributed arrays `g_num_pp`, `p_g_co_pp` and `g_g_pp` replace their serial counterparts `g_num`, `g_coord` and `g_g`. Loop `elements_2` with embedded loop `integrating_points_1` clearly carries over although the parallel version assumes uniform element properties throughout the mesh. In all other aspects the serial and parallel programs are essentially identical although of course the serial and parallel solution algorithm libraries are not interchangeable. Data are listed as Figure 12.35 involving about a quarter of a million equations with results as Figure 12.36 and some performance statistics as Figure 12.37. For larger data sets, Figures 12.38 and 12.39 show performance measured on an SGI Origin 3000 computer.

```
nels    nxe  nze  nip
78125   25   25    8

aa     bb      cc       rho
0.04  0.04   0.04      1.0

e    v
1.0  0.3

nmodes
5

el   er
0.0  1.0

lalfa  leig  lx       lz    acc
5000    20   100      5000  1.0e-9
```

Figure 12.35 Data for Program 12.8 example

```
This job ran on 32   processors

There are    85176 nodes     676 restrained  and  253500 equations
 Time after setup is  : 1.52999999999883585
The range is  0.0000E+00  to   0.1000E+01
There are        9 eigenvalues in the range
It took     1005  iterations
The eigenvalues are   :
  0.1580E-02  0.3217E-01  0.4592E-01  0.9977E-01  0.2610E+00  0.2894E+00
  0.7217E+00  0.8033E+00  0.8873E+00
The eigenvectors are  :
Eigenvector number    1 is:
  0.6026E+00  0.6025E+00 -0.6662E-01  0.6026E+00  0.6025E+00 -0.7326E-01
Eigenvector number    2 is:
  0.9296E-01  0.6500E+00 -0.8792E-04  0.3099E-01  0.6500E+00  0.8798E-04
Eigenvector number    3 is:
 -0.5045E+00 -0.5034E+00  0.2030E+00 -0.5045E+00 -0.5034E+00  0.2231E+00
Eigenvector number    4 is:
 -0.9671E-03  0.4628E-04  0.6334E+00 -0.9672E-03 -0.4628E-04  0.6334E+00
Eigenvector number    5 is:
  0.4333E+00  0.4247E+00 -0.2817E+00  0.4335E+00  0.4250E+00 -0.3083E+00
 This analysis took   : 85.7199999999975262
```

Figure 12.36 Results from Program 12.8 example

Mesh	**No of Processors**	**Analysis Time(secs)[Using BLAS]**
25 x 125 x 25	16	163.8
	32	85.7 [22]

Figure 12.37 Performance statistics: Program 12.8 (IBM SP2)

320,000 elements, 1.1M equations

# Processors	16	64	256	384
Total	327.4	70.5	21.3	17.4
Setup	52.0	12.9	4.5	4.9
Iterations	1,609	1,636	1,629	1,663

Figure 12.38 Larger data set: Program 12.8 (SGI Origin 3000)

2.56M elements, 7.8M equations

# Processors	<186 insufficient memory	192	256	384
Total		1,636.8	414.2	252.1
Setup		141.6	108.8	72.4
Iterations		3,174	3,198	3,254

Figure 12.39 Larger data set: Program 12.8 (SGI Origin 3000)

Program 12.9 Forced vibration analysis of a three-dimensional elastic solid. Implicit integration in time. Compare Program 11.4.

```
 PROGRAM p129
!---------------------------------------------------------------------------
!      Program 11.6 forced vibration of a 3 - d  elastic
!      solid  using uniform 20-node hexahedral elements  (nxe even)
!      numbered in the x-z direction - lumped or consistent mass
!      implicit integration by theta method : parallel version
!---------------------------------------------------------------------------
 USE new_library;   USE  geometry_lib ; USE precision; USE gather_scatter6
 USE global_variables1;USE timing;USE mp_module;USE utility; IMPLICIT NONE
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=3,nod=20,nst=6,neq_temp,nn_temp,    &
          i,j,k,iel,ndim=3,nstep,npri,iters,limit , it,is  ,nres
! ndof, nels, ntot, neq are now global variables ; not declared
 REAL(iwp)::aa,bb,cc,e,v,det,rho,alpha1,beta1,omega,theta,period,pi,dtim, &
            volume,c1,c2,c3,c4,real_time,tol,big,up,alpha,beta
 CHARACTER(LEN=15)::element='hexahedron'
 LOGICAL :: consistent = .FALSE.  , converged
!---------------------------- dynamic arrays-----------------------------
 REAL(iwp),ALLOCATABLE::loads_pp(:),points(:,:),dee(:,:),coord(:,:),      &
                        fun(:),jac(:,:), der(:,:),deriv(:,:), weights(:),    &
                        bee(:,:),km(:,:),p_g_co_pp(:,:,:),x1_pp(:),          &
                        d1x1_pp(:),d2x1_pp(:),emm(:,:),ecm(:,:),x0_pp(:),    &
                        d1x0_pp(:),d2x0_pp(:),store_km_pp(:,:,:),vu_pp(:),   &
                        store_mm_pp(:,:,:),u_pp(:),p_pp(:),d_pp(:),          &
                        x_pp(:),xnew_pp(:),pmul_pp(:,:),utemp_pp(:,:),       &
                        diag_precon_pp(:),diag_precon_tmp(:,:),temp_pp(:,:,:)
 INTEGER, ALLOCATABLE::rest(:,:), g(:), num(:), g_num_pp(:,:), g_g_pp(:,:)
!-----------------------input and initialisation------------------------
 timest(1) = elap_time( ) ;   CALL find_pe_procs(numpe,npes)
 IF(numpe==npes) THEN
  OPEN (10,FILE='p129.dat',STATUS=    'OLD',ACTION='READ')
  READ (10,*) nels,nxe,nze,nip,aa,bb,cc,rho,e,v,                          &
              alpha1,beta1,nstep,npri,theta,omega,tol,limit
 END IF
 CALL bcast_inputdata_p129(numpe,npes,nels,nxe,nze,nip,aa,bb,cc,rho,e,v,  &
                           alpha1,beta1,nstep,theta,npri,omega,tol,limit)
 CALL calc_nels_pp  ; ndof=nod*nodof ; neq_temp = 0  ;  nn_temp = 0
 ntot = ndof; nye = nels/nxe/nze; nr=3*nxe*nze+2*nxe+2*nze+1
 nres = 3*(nye*(nxe+1)*(nze+1)+nr*(nye-1)+(nxe+1))
 ALLOCATE(rest(nr,nodof+1),points(nip,ndim),g(ntot),fun(nod),             &
            dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),     &
            der(ndim,nod), deriv(ndim,nod), bee(nst,ntot), km(ntot,ntot), &
            num(nod),g_num_pp(nod,nels_pp),g_g_pp(ntot,nels_pp),          &
            emm(ntot,ntot),ecm(ntot,ntot),p_g_co_pp(nod,ndim,nels_pp),    &
            store_km_pp(ntot,ntot,nels_pp),utemp_pp(ntot,nels_pp),        &
            pmul_pp(ntot,nels_pp),store_mm_pp(ntot,ntot,nels_pp),         &
            temp_pp(ntot,ntot,nels_pp),diag_precon_tmp(ntot,nels_pp))
  rest = 0; DO i=1,nr; rest(i,1) = i; END DO ; ielpe = iel_start
  pi=ACOS(-1._iwp)   ; period = 2._iwp*pi/omega ; dtim =period/20._iwp
  c1=(1._iwp-theta)*dtim; c2=beta1-c1
  c3=alpha1+1._iwp/(theta*dtim); c4=beta1+theta*dtim
  CALL deemat (dee,e,v); CALL sample(element,points,weights)
!-------------- loop the elements to set up global arrays ----------------
  elements_1: DO iel = 1 , nels_pp
              CALL geometry_20bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
              CALL find_g( num , g , rest ) ;  g_num_pp(:,iel) = num
```

```
             p_g_co_pp(:,:,iel) = coord; g_g_pp(:,iel)=g; ielpe=ielpe+1
             i = MAXVAL(g);    j = MAXVAL(num)
             IF(i>neq_temp)neq_temp = i ; IF(j>nn_temp)nn_temp = j
  END DO elements_1
  neq = reduce(neq_temp);  nn = reduce(nn_temp)
  CALL calc_neq_pp          ;   CALL make_ggl(g_g_pp)
  DO i=1,neq_pp; IF(nres==ieq_start+i-1) THEN; it=numpe; is = i; END IF
  END DO
  IF(numpe==it) THEN
    OPEN (11,FILE='p129.res',STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I5,A)') "This job ran on  ", npes, "  processors"
    WRITE(11,'(A)') "Global coordinates and node numbers "
    DO i= 1, nels_pp , nels_pp - 1
           WRITE(11,'(A,I8)')"Element ",i; num = g_num_pp(:,i)
      DO k = 1,nod;WRITE(11,'(A,I8,3E12.4)')                             &
        "  Node",num(k),p_g_co_pp(k,:,i); END DO
    END DO
    WRITE(11,'(A,3(I8,A))') "There are ",nn," nodes",nr," restrained and",&
                           neq," equations"
    WRITE(11,*) "Time after setup is  :", elap_time( ) - timest(1)
  END IF
  ALLOCATE(x0_pp(neq_pp),d1x0_pp(neq_pp),x1_pp(neq_pp),vu_pp(neq_pp),     &
         diag_precon_pp(neq_pp),u_pp(neq_pp),                             &
         d2x0_pp(neq_pp),loads_pp(neq_pp),d1x1_pp(neq_pp),d2x1_pp(neq_pp),&
         d_pp(neq_pp),p_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp))
    xnew_pp=.0_iwp; p_pp=.0_iwp; diag_precon_pp=.0_iwp
    store_km_pp=.0_iwp;store_mm_pp=.0_iwp;diag_precon_tmp = .0_iwp
!-- element stiffness and mass integration ,storage and preconditioner --
 elements_2: DO iel = 1 , nels_pp
             coord=p_g_co_pp(:,:,iel);km=.0_iwp;volume=.0_iwp; emm=.0_iwp
         gauss_points_1: DO i = 1 , nip
            CALL shape_der (der,points,i) ; jac = MATMUL(der,coord)
            det = determinant(jac)  ; CALL invert(jac)
            deriv = matmul(jac,der) ; CALL beemat (bee,deriv)
            km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee) *det* weights(i)
            volume=volume + det*weights(i); CALL shape_fun(fun,points,i)
            IF(consistent) THEN
             CALL ecmat(ecm,fun,ntot,nodof); ecm=ecm*det*weights(i)*rho
             emm = emm + ecm
            END IF
         END DO gauss_points_1
      IF(.NOT.consistent) THEN
        DO i=1,ntot; emm(i,i)=volume*rho/13._iwp; END DO
        DO i=1,19,6 ; emm(i,i)=emm(4,4)*.125_iwp; END DO
        DO i=2,20,6 ; emm(i,i)=emm(4,4)*.125_iwp; END DO
        DO i=3,21,6 ; emm(i,i)=emm(4,4)*.125_iwp; END DO
        DO i=37,55,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
        DO i=38,56,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
        DO i=39,57,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
      END IF
   store_km_pp (: , : , iel ) = km ; store_mm_pp( : , : , iel ) = emm
  DO k=1,ntot
   diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)+emm(k,k)*c3+km(k,k)*c4
  END DO
 END DO elements_2    ; CALL scatter(diag_precon_pp,diag_precon_tmp)
 diag_precon_pp = 1._iwp / diag_precon_pp; DEALLOCATE(diag_precon_tmp)
!----------------------initial conditions -----------------------------
           x0_pp = .0_iwp; d1x0_pp = .0_iwp; d2x0_pp = .0_iwp
```

```
!----------------------- time stepping loop -----------------------------
   real_time = .0_iwp
   IF(numpe==it) THEN
     WRITE(11,'(A)') "    Time t  cos(omega*t) Displacement Iterations"
   END IF
  timesteps: DO j = 1 , nstep
             real_time = real_time + dtim   ; loads_pp = .0_iwp
    u_pp = .0_iwp   ;  vu_pp = .0_iwp
     elements_3: DO iel = 1 , nels_pp    ! gather for rhs multiply
        temp_pp(:,:,iel)=store_km_pp(:,:,iel)*c2+ store_mm_pp(:,:,iel)*c3
     END DO elements_3
     CALL gather(x0_pp,pmul_pp)
     DO iel=1,nels_pp
            utemp_pp(:,iel) = MATMUL(temp_pp(:,:,iel),pmul_pp(:,iel))
     END DO ; CALL scatter(u_pp,utemp_pp)
!-------------------- Velocity Bit --------------------------------------
     temp_pp = store_mm_pp/theta; CALL gather(d1x0_pp,pmul_pp)
     DO iel = 1,nels_pp
           utemp_pp(:,iel)=MATMUL(temp_pp(:,:,iel),pmul_pp(:,iel))
     END DO  ;     CALL scatter(vu_pp,utemp_pp) ! doesn't add to last u_pp
     IF(numpe==it) THEN
          loads_pp(is)=theta*dtim*cos(omega*real_time)+        &
                           c1*cos(omega*(real_time-dtim)) !
     END IF
     loads_pp = u_pp + vu_pp + loads_pp
!------------------ solve simultaneous equations by pcg ------------------
     d_pp = diag_precon_pp*loads_pp; p_pp = d_pp; x_pp = .0_iwp
     iters = 0
     iterations  :      DO
             iters = iters + 1   ;  u_pp = 0._iwp  ; vu_pp = .0_iwp
           temp_pp=store_mm_pp*c3+store_km_pp*c4;CALL gather(p_pp,pmul_pp)
           elements_4 : DO iel = 1, nels_pp
              utemp_pp(:,iel) = MATMUL(temp_pp(:,:,iel),pmul_pp(:,iel))
           END DO elements_4 ;         CALL scatter(u_pp,utemp_pp)
!---------------------------pcg equation solution------------------------
         up=DOT_PRODUCT_P(loads_pp,d_pp);alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
         xnew_pp = x_pp + p_pp* alpha ; loads_pp=loads_pp - u_pp*alpha
         d_pp = diag_precon_pp*loads_pp  !
         beta=DOT_PRODUCT_P(loads_pp,d_pp)/up; p_pp=d_pp+p_pp*beta
         u_pp = xnew_pp
         CALL checon_par(xnew_pp,x_pp,tol,converged,neq_pp)
         IF(converged .OR. iters==limit) EXIT
     END DO iterations
       x1_pp=xnew_pp
       d1x1_pp=(x1_pp-x0_pp)/(theta*dtim)-d1x0_pp*(1._iwp-theta)/theta
       d2x1_pp=(d1x1_pp-d1x0_pp)/(theta*dtim)-d2x0_pp*(1._iwp-theta)/theta
          IF(j/npri*npri==j .AND. numpe==it)   WRITE(11,'(3E12.4,I10)')   &
            real_time,cos(omega*real_time),x1_pp(is),iters
            x0_pp = x1_pp; d1x0_pp = d1x1_pp; d2x0_pp = d2x1_pp
  END DO timesteps
  IF(numpe==it) WRITE(11,*)"This analysis took   :", elap_time()-timest(1)
  CALL shutdown( )
END PROGRAM p129
```

New scalar reals:

`alpha1`	Rayleigh damping parameter
`beta1`	Rayleigh damping parameter

`c1`	
`c2`	intermediate reals
`c3`	
`c4`	
`period`	period of oscillation
`volume`	element volume

New scalar logicals:

`consistent`	`.TRUE.` if element mass is "consistent"

New dynamic real arrays:

`d1x0_pp`	distributed old velocity vector
`d2x0_pp`	distributed old acceleration vector
`d1x1_pp`	distributed new velocity vector
`d2x1_pp`	distributed new acceleration vector
`store_mm_pp`	distributed element mass storage
`temp_pp`	distributed intermediate vector
`vu_pp`	distributed intermediate vector
`x0_pp`	distributed old displacement vector
`x1_pp`	distributed new displacement vector

The now familiar consistency of loops `elements_1` and `elements_2` with embedded `gauss_pts_1` appears again. The parallel (3D) version has the lumped mass matrix for 20-node elements hand-coded in place of the `elmat` routine used in the serial (2D) case which used 8-node elements. The time stepping loop involves comparable loop `elements_3` while the pcg section involves comparable loop `elements_4`. Data are listed as Figure 12.40 with results as Figure 12.41 and performance statistics as Figure 12.42.

```
nels   nxe  nze  nip
2560    8    8   27

aa      bb      cc      rho
0.125  0.125   0.125   1.0

e    v
1.0  0.3

alpha1  beta1
0.0008  0.5

nstep  npri  theta
32      1    1.0

omega   tol      limit
0.01   0.0001   3000
```

Figure 12.40 Data for Program 12.9 example

```
This job ran on    32  processors

There are    12465 nodes     225 restrained and   36720 equations
 Time after setup is  : 0.339999999996507540
    Time t  cos(omega*t) Displacement Iterations
  0.3142E+02  0.9511E+00  0.3469E+03       428
  0.6283E+02  0.8090E+00  0.5158E+03       420
  0.9425E+02  0.5878E+00  0.4672E+03       419
.
.
.
  0.9425E+03 -0.1000E+01 -0.5935E+03       421
  0.9739E+03 -0.9511E+00 -0.5696E+03       421
  0.1005E+04 -0.8090E+00 -0.4899E+03       422
 This analysis took   : 562.680000000000291
```

Figure 12.41 Results from Program 12.9 example

Mesh	No of Processors	Analysis Time(secs)
8x40x8	16	1601
	32	562.7

Figure 12.42 Performance statistics: Program 12.9 (IBM SP2)

Program 12.10 Forced vibration analysis of three-dimensional elasto-plastic solid. Explicit integration in time. Compare Program 11.5.

```
 PROGRAM p1210
!-----------------------------------------------------------------------
!  Program 11.7 forced vibration of an elastic-plastic(Von Mises) solid
!  using 20-node hexahedral elements; nxe even; viscoplastic strain method
!  regular box mesh : lumped mass, explicit integration: parallel version
!-----------------------------------------------------------------------
 USE new_library; USE geometry_lib; USE mp_module; USE timing; USE utility
 USE global_variables1; USE precision;  USE gather_scatter6; IMPLICIT NONE
 ! ndof,nels,neq,ntot are now global variables - not declared
 INTEGER::nxe,nye,nze,nn,nr,nip,nodof=3,nod=20,nst=6,loaded_nodes,nres,   &
         i,j,k,jj,iel,ndim=3,nstep ,npri,num_no,no_index_start ,          &
         neq_temp,nn_temp, is,it
 REAL(iwp)   ::aa,bb,cc,rho,dtim,e,v,det,sbary,pload,sigm,f,fnew,fac,     &
              volume,sbar,dsbar,lode_theta,real_time
 CHARACTER (LEN = 15) :: element = 'hexahedron'
!---------------------------- dynamic arrays----------------------------
 REAL(iwp), ALLOCATABLE :: points(:,:),bdylds_pp(:),x1_pp(:),d1x1_pp(:),  &
                   stress(:),pl(:,:),emm(:),d2x1_pp(:),tensor_pp(:,:,:), &
                   etensor_pp(:,:,:),val(:),dee(:,:),coord(:,:),mm_pp(:),&
                   jac(:,:),weights(:), der(:,:),deriv(:,:),bee(:,:),    &
                   eld(:),eps(:),sigma(:),bload(:),eload(:),mm_tmp(:,:), &
                   p_g_co_pp(:,:,:),pmul_pp(:,:),utemp_pp(:,:)
 INTEGER, ALLOCATABLE :: rest(:,:), g(:), no(:), num(:),no_local(:),      &
                         g_num_pp(:,:), g_g_pp(:,:),no_local_temp(:)
!----------------------input and initialisation-------------------------
 timest(1) = elap_time( )   ;  CALL find_pe_procs(numpe,npes)
 IF(numpe==npes) THEN
  OPEN (10,FILE='p1210.dat',STATUS=    'OLD',ACTION='READ')
  READ (10,*) aa,bb,cc,sbary,e,v,rho,pload,   &
              nels,nxe,nze,nip,loaded_nodes,dtim,nstep,npri
 END IF
 CALL bcast_inputdata_p1210(numpe,npes,nels,nxe,nze,nip,loaded_nodes,     &
                            aa,bb,cc,rho,e,v,sbary,pload,nstep,dtim,npri)
 CALL calc_nels_pp;   ndof=nod*nodof   ; ntot = ndof ; neq_temp = 0
 nn_temp = 0 ;nr = 3*nxe*nze+2*nxe+2*nze+1 ; nye = nels/nxe/nze
 nres = 3*(nye*(nxe+1)*(nze+1)+nr*(nye-1)+(nxe+1))
 ALLOCATE (rest(nr,nodof+1), points(nip,ndim),weights(nip),num(nod),      &
           p_g_co_pp(nod,ndim,nels_pp),dee(nst,nst),coord(nod,ndim),      &
           tensor_pp(nst,nip,nels_pp),no(loaded_nodes), pl(nst,nst),      &
           etensor_pp(nst,nip,nels_pp),jac(ndim,ndim),der(ndim,nod),      &
           deriv(ndim,nod),g_num_pp(nod,nels_pp),bee(nst,ntot),eld(ntot),&
           eps(nst),sigma(nst),emm(ntot),bload(ntot),eload(ntot),g(ntot),&
           stress(nst),val(loaded_nodes),g_g_pp(ntot,nels_pp),            &
           mm_tmp(ntot,nels_pp),pmul_pp(ntot,nels_pp),                    &
           utemp_pp(ntot,nels_pp),no_local_temp(loaded_nodes))
  tensor_pp=.0_iwp; etensor_pp=.0_iwp; mm_tmp=.0_iwp; pmul_pp=.0_iwp
  utemp_pp=.0_iwp
  rest = 0; DO i=1,nr; rest(i,1) = i; END DO   ; ielpe = iel_start
  no = nres;    val = pload
!----------- loop the elements to set up global arrays -------------------
      elements_0:DO iel = 1 , nels_pp
                  CALL geometry_20bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
                  CALL find_g(num,g,rest)  ; g_num_pp(:,iel) = num
                  i = MAXVAL(g) ; j = MAXVAL(num)
                  IF(i > neq_temp) neq_temp=i; IF(j > nn_temp)nn_temp = j
                  p_g_co_pp(:,:,iel)=coord; g_g_pp(:,iel)=g;ielpe=ielpe+1
      end do elements_0
 neq = reduce(neq_temp);  nn = reduce(nn_temp)
 CALL calc_neq_pp ;  CALL make_ggl(g_g_pp)
 DO i = 1 , neq_pp; IF(nres==ieq_start+i-1) THEN; it=numpe; is=i; END IF
 END DO
 IF(numpe==it) THEN
   OPEN (11,FILE='p1210.res',STATUS='REPLACE',ACTION='WRITE')
   WRITE(11,'(A,I5,A)') "This job ran on ",npes, " processors"
   WRITE(11,'(A)') "Global coordinates and node numbers "
   DO i= 1, nels_pp , nels_pp - 1
             WRITE(11,'(A,I8)')"Element ",i  ; num = g_num_pp(:,i)
     DO k = 1,nod;WRITE(11,'(A,I8,3E12.4)')                               &
       "  Node",num(k),p_g_co_pp(k,:,i); END DO
   END DO
   WRITE(11,'(A,3(I8,A))') "There are ",nn," nodes",nr," restrained and",&
                           neq," equations"
   WRITE(11,*) "Time after setup is  :", elap_time( ) - timest(1)
 END IF
  ALLOCATE(bdylds_pp(neq_pp),x1_pp(neq_pp),d1x1_pp(neq_pp),               &
            d2x1_pp(neq_pp),mm_pp(neq_pp))
  bdylds_pp=.0_iwp; x1_pp=0.0_iwp; d1x1_pp=0.0_iwp
  d2x1_pp=0.0_iwp; mm_pp=0.0_iwp ;  CALL sample(element,points,weights)
!--------------------calculate diagonal mass matrix ----------------------
 elements_1: DO iel = 1 , nels_pp
              coord = p_g_co_pp(:,:,iel); volume= .0_iwp
           gauss_pts_1: DO i = 1 , nip
               CALL shape_der (der,points,i); jac = MATMUL(der,coord)
               det = determinant(jac); volume=volume+det*weights(i)*rho
           END DO gauss_pts_1
               emm=volume/13._iwp;emm(1:19:6)=emm(4)*.125_iwp
               emm(2:20:6)=emm(4)*.125_iwp; emm(3:21:6)=emm(4)*.125_iwp
               emm(37:55:6)=emm(4)*.125_iwp;emm(38:56:6)=emm(4)*.125_iwp
               emm(39:57:6)=emm(4)*.125_iwp
           mm_tmp(:,iel)=mm_tmp(:,iel) + emm
 END DO elements_1
 CALL scatter(mm_pp,mm_tmp); DEALLOCATE(mm_tmp)
!------------------------- reindexing information -----------------------
  CALL reindex_fixed_nodes                                               &
              (ieq_start,no,no_local_temp,num_no,no_index_start)
  ALLOCATE(no_local(1:num_no)) ; no_local = no_local_temp(1:num_no)
  DEALLOCATE (no_local_temp)
!--------------------- explicit integration loop -------------------------
 real_time = .0_iwp
 IF(numpe==it) THEN
  WRITE(11,'(A)') "   Time     Displacement  Velocity   Acceleration"
  WRITE(11,'(4e12.4)')real_time,x1_pp(is),d1x1_pp(is),d2x1_pp(is)
 END IF
 time_steps : DO jj = 1 , nstep
!-------------------------  apply the load ------------------------------
 real_time = real_time + dtim
 x1_pp = x1_pp +(d1x1_pp+d2x1_pp*dtim*.5_iwp)*dtim  ;  bdylds_pp = .0_iwp
!-------------------- element stress-strain relationship -----------------
 pmul_pp = .0_iwp; utemp_pp = .0_iwp; CALL gather(x1_pp,pmul_pp)
 elements_2: DO iel = 1 , nels_pp
             coord=p_g_co_pp(:,:,iel); bload=.0_iwp; eld = pmul_pp(:,iel)
            gauss_pts_2:  DO i =1 , nip; dee=.0_iwp; CALL deemat(dee,e,v)
              CALL shape_der (der,points,i);  jac = MATMUL(der,coord)
              det = determinant(jac)  ;   CALL invert(jac)
              deriv = MATMUL(jac,der) ;  CALL beemat (bee,deriv)
              eps=MATMUL(bee , eld) ; eps = eps - etensor_pp(:, i , iel )
              sigma= MATMUL( dee , eps ); stress=sigma+tensor_pp(:,i,iel)
              CALL invar(stress,sigm,dsbar,lode_theta);fnew = dsbar-sbary
!---------------------- check whether yield is violated ------------------
               IF(fnew>=.0_iwp) THEN
                 stress= tensor_pp(:,i,iel)
                 CALL invar(stress,sigm,sbar,lode_theta)
                 f = sbar - sbary; fac = fnew/(fnew - f)
                 stress = tensor_pp ( : , i , iel )+(1._iwp-fac) * sigma
                 CALL vmpl(e,v,stress,pl); dee = dee - fac * pl
               END IF
              sigma=MATMUL(dee ,eps); sigma=sigma+tensor_pp(: , i , iel )
              eload=MATMUL(sigma,bee); bload=bload+eload*det * weights(i)
!-----------------------update Gauss point stresses and strains ----------
              tensor_pp( : , i , iel) = sigma
              etensor_pp( : , i , iel ) = etensor_pp( : , i , iel ) + eps
            END DO gauss_pts_2
            utemp_pp(:,iel) = utemp_pp(:,iel) - bload
 END DO elements_2
 CALL scatter (bdylds_pp,utemp_pp)
      DO i = 1 , num_no
         j = no_local(i) - ieq_start + 1
         bdylds_pp(j) = bdylds_pp(j)+val(no_index_start + i - 1) * pload
      END DO
      bdylds_pp = bdylds_pp / mm_pp
      d1x1_pp=d1x1_pp+(d2x1_pp+bdylds_pp)*.5_iwp*dtim; d2x1_pp = bdylds_pp
      IF(jj==jj/npri*npri .AND. numpe==it) WRITE(11,'(4E12.4)')real_time, &
```

```
                                                  x1_pp(is),d1x1_pp(is),d2x1_pp(is)
END DO time_steps
IF(numpe==it) WRITE(11,*) "This analysis took  :", elap_time( )-timest(1)
CALL shutdown( )
END PROGRAM p1210
```

New scalar integers:

`jj`	simple counter
`loaded_nodes`	number of loaded nodes

New scalar reals:

`fac`	yield factor
`fnew`	new yield function
`pload`	load multiple
`sbar`	shear stress
`sbary`	shear yield stress

New dynamic real arrays:

`etensor_pp`	distributed element strains
`mm_pp`	distributed mass vector
`mm_tmp`	temporary vector
`pl`	plasticity matrix

In the parallel version, an extra loop, `elements_0`, creates geometry as is done by `elements_1` in serial. Then parallel `elements_1` creates the lumped mass matrix for 20-node hexahedral elements. Following this, loops `elements_2` in both programs are equivalent. Uniform elements are assumed in 3D. Data are listed as Figure 12.43 with results as Figure 12.44 and performance statistics as Figure 12.45

```
aa    bb    cc    sbary
0.1   0.1   0.1   1.0e8

e     v    rho
1.0  0.3   1.0

pload
-1.0

nels  nxe  nze  nip
5000  10   10   27

loaded_nodes
1

dtim   nstep   npri
0.005  1000    100
```

Figure 12.43 Data for Program 12.10 example

```
This job ran on   32 processors

There are    23441 nodes     341 restrained and   69300 equations
 Time after setup is  : 0.289999999993597157
   Time     Displacement  Velocity   Acceleration
  0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
  0.5000E+00  0.7171E+02 -0.1272E+03 -0.1294E+04
  0.1000E+01  0.7047E+02 -0.7126E+02  0.1556E+04
  0.1500E+01  0.7305E+02 -0.8662E+00 -0.2327E+03
  0.2000E+01  0.7149E+02  0.1814E+02  0.1927E+04
  0.2500E+01  0.7584E+02 -0.3821E+02 -0.2451E+03
  0.3000E+01  0.7851E+02 -0.1181E+02  0.5939E+03
  0.3500E+01  0.8550E+02 -0.1367E+02 -0.1659E+04
  0.4000E+01  0.8610E+02 -0.3553E+02  0.4863E+03
  0.4500E+01  0.8925E+02  0.1898E+02 -0.5803E+03
  0.5000E+01  0.9213E+02 -0.2836E+01 -0.5797E+03
 This analysis took  : 707.939999999995052
```

Figure 12.44 Results from Program 12.10 example

Mesh	No of Processors	Analysis Time(secs)
10x50x10	16	1392
	32	708

Figure 12.45 Performance statistics: Program 12.10 (IBM SP2)

12.3 Performance data for a "Beowulf" PC cluster

The results, which follow in Figures 12.46 and 12.47, show speed-up performance for Programs 12.1 and 12.3 respectively on a low-cost cluster of about 8 standard PCs at the Department of Civil Engineering, University of Madrid.

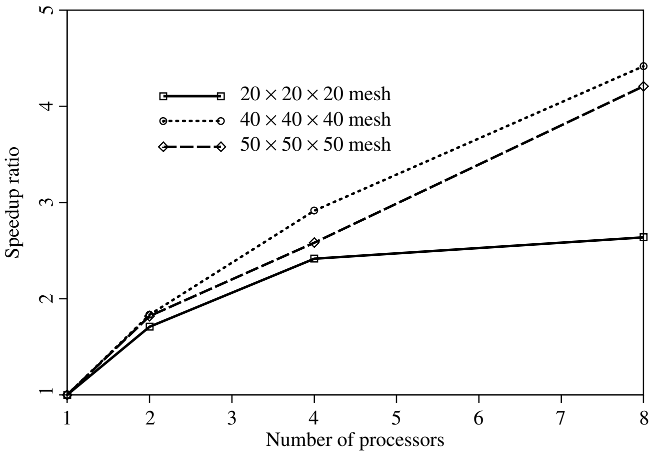


Figure 12.46 Performance data: Program 12.1 (MARIAN Cluster)

The results are qualitatively similar to those given by Smith (2000) for a different PC cluster. It seems that substantial gains in processing speed can be achieved for up to at least 10 processors.

Again, substantial savings in analysis time can be achieved for modest numbers of processors. Perhaps more importantly is the ease with which problems involving millions of unknowns can be successfully tackled using such low cost hardware.

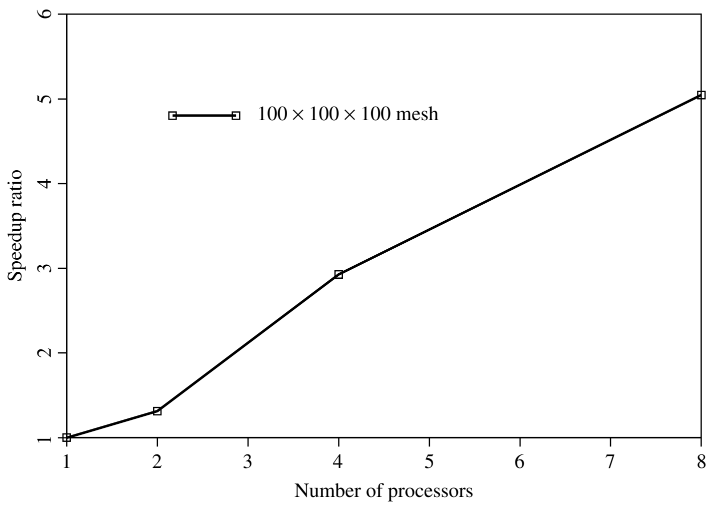


Figure 12.47 Performance data: Program 12.3 (MARIAN Cluster)

12.4 Conclusions

All the serial program types described in Chapters 5 to 11 have been parallelised using a consistent strategy. Performance data are excellent for a certain genre of current "supercomputer", circa 2002. Analyses have "scaled" well for up to about 50 million elements and about 500 processors. For lower cost systems, typified by "Beowulf" PC clusters, the ratio of processing to communication speed is far higher—indeed many PCs may have faster processors than a supercomputer. This seems to limit "scalability" on a current cluster with simple Ethernet communication to about 10 PCs. Nevertheless very large problems with millions of elements have been successfully solved on such a cluster. A current trend is to produce a new range of "supercomputer" with faster processors but essentially unchanged communication capabilities. In that case we can anticipate a deterioration in scalability and there is scope for improving the gather/scatter routines used herein and for developing iterative algorithms in which potential communication is minimised. These developments would be of benefit both to "supercomputers" and to "clusters".

Glossary of variable names used in Chapter 12

Scalar integers:

`cjiters`	conjugate gradient iteration counter
`cjits`	conjugate gradient iteration ceiling
`cjtot`	total number of conjugate gradient iterations
`cj_tot`	total number of BiCGStab iterations
`ell`	l in the BiCGStab(l) process
`fixed_freedoms`	number of fixed freedoms
`fixed_nodes`	number of fixed nodes
`i`	simple counter

`iel`	element counter
`iflag`	failure flag
`incs`	number of load increments
`is`	location of desired output freedom
`it`	processor on which desired output resides
`iters`	iteration counter
`iy`	simple counter
`j`	simple counter
`jflag`	failure flag
`jj`	simple counter
`k`	simple counter
`l`	simple counter
`lalfa`	length of `alfa` array
`leig`	length of `eig` array
`limit`	iteration ceiling
`loaded_freedoms`	number of loaded freedoms
`loaded_nodes`	number of loaded nodes
`lp`	output channel number
`lx`	problem dependent array size
`lz`	problem dependent array size
`ndim`	number of dimensions
`neig`	problem dependent array size
`neq_temp`	temporary equation sum
`nip`	number of integrating points
`nle`	number of loaded elements (side of square)
`nmodes`	number of eigenmodes computed
`nn`	number of nodes in the mesh
`nn_temp`	temporary node sum
`nod`	number of nodes per element
`nodf`	number of pressure degrees of freedom per element
`nodof`	number of degrees of freedom per node
`no_index_start`	start address of loaded/fixed freedoms
`npri`	print interval
`nr`	number of restrained nodes
`nres`	number of output freedom
`ns`	timestep counter
`nst`	number of stress–strain terms
`nstep`	number of timesteps in analysis
`num_no`	number of processors holding load/displacement data
`nxe`	number of elements in x direction
`nye`	number of elements in y direction
`nze`	number of elements in z direction
`n_t`	total number of degrees of freedom per element
`plasiters`	plastic iteration counter
`plasits`	plastic iteration ceiling

Scalar reals:

`aa`	*x* dimension of elements
`acc`	accuracy parameter
`alpha`	pcg parameter
`alpha1`	Rayleigh damping parameter
`bb`	*y* dimension of elements
`beta`	pcg parameter
`beta1`	Rayleigh damping parameter
`big`	largest component of a vector
`c`	cohesion
`cc`	*z* dimension of elements
`cjtol`	conjugate gradient iteration tolerance
`cons`	consolidation pressure
`c1`	
`c2`	intermediate reals
`c3`	
`c4`	
`det`	determinant of Jacobian matrix
`dq1`	Mohr–Coulomb plastic potential derivative
`dq2`	Mohr–Coulomb plastic potential derivative
`dq3`	Mohr–Coulomb plastic potential derivative
`dsbar`	shear stress invariant
`dt`	viscoplastic “time” step
`dtim`	timestep
`e`	Young’s Modulus
`el`	left limit of eigenvalue spectrum
`er`	right limit of eigenvalue spectrum
`error`	residual error
`f`	current stress state
`fac`	yield factor
`fnew`	new yield function
`gama`	intermediate value
`kappa`	`kappa` in BiCGStab process
`kx`	conductivity in *x*-direction
`ky`	conductivity in *y*-direction
`kz`	conductivity in *z*-direction
`lode_theta`	Lode angle
`norm_r`	norm of residual
`omega`	intermediate value
`penalty`	value of penalty restraint
`period`	period of oscillation
`phi`	angle of internal friction
`plastol`	plastic iteration tolerance
`pload`	load multiple
`pp`	intermediate value
`presc`	prescribed value of load/displacement

`psi`	angle of dilation
`q`	total load
`real_time`	accumulated time
`rho`	density
`rho1`	intermediate value
`r0_norm`	starting residual norm
`sbar`	shear stress
`sbary`	shear yield stress
`sigm`	mean stress invariant
`snph`	sine of angle of internal friction
`theta`	parameter in "theta" integrator
`tol`	convergence tolerance
`ubar`	average x velocity
`up`	pcg parameter
`v`	Poisson's Ratio
`val0`	initial value
`vbar`	average y velocity
`visc`	viscosity
`volume`	element volume
`wbar`	average z velocity
`x0`	start value

Scalar characters:

`element`	element type

Scalar logicals:

`cj_converged`	set to `.TRUE.` if conjugate gradient iterations converged
`consistent`	set to `.TRUE.` if element mass is "consistent"
`converged`	set to `.TRUE.` if solution converged
`plastic_converged`	set to `.TRUE.` if plastic iterations converged

Dynamic integer arrays:

`g`	element steering vector
`g_g_pp`	distributed global steering matrix
`g_num_pp`	distributed global element node numbers matrix
`g_t`	total element g vector
`jeig`	intermediate array
`no`	freedoms to be loaded/fixed
`no_f`	vector of fixed freedom numbers
`no_local`	local (processor) freedoms
`no_local_temp`	temporary store
`no_local_temp_f`	temporary vector
`nu`	intermediate array
`num`	element node numbers
`rest`	node freedom restraints

Dynamic real arrays:

`alfa`	intermediate array

`ans_pp`	distributed answer vector
`bdylds_pp`	distributed body loads vector
`bee`	strain-displacement matrix
`beta`	intermediate array
`bload`	element body loads
`b_pp`	distributed right hand side vector
`c`	coupling matrix
`col`	column array
`coord`	element nodal coordinates
`coordf`	nodal coordinates of pressure nodes
`c11`	
`c12`	
`c21`	
`c23`	c arrays—see equation (2.115)
`c32`	
`c24`	
`c42`	
`dee`	stress–strain matrix
`del`	intermediate array
`der`	derivatives wrt local coordinates
`derf`	local derivatives of pressure shape functions
`deriv`	derivatives wrt global coordinates
`derivf`	global derivatives of pressure shape functions
`devp`	increment of viscoplastic strain
`diag_pp`	distributed diagonal vector
`diag_precon_pp`	distributed diagonal preconditioning matrix
`diag_precon_tmp`	temporary store
`diag_tmp`	diagonal mass matrix
`d1x0_pp`	distributed old velocity vector
`d1x1_pp`	distributed new velocity vector
`d2x0_pp`	distributed old acceleration vector
`d2x1_pp`	distributed new acceleration vector
`d_pp`	distributed pcg vector
`ecm`	element consistent mass matrix
`eig`	intermediate array
`eld`	element nodal displacements
`eld_pp`	distributed nodal displacements
`eload`	accumulating element body loads
`emm`	accumulated element mass matrix
`eps`	element strains
`erate`	viscoplastic strain rate
`etensor_pp`	distributed element strains
`evp`	viscoplastic strains
`evpt_pp`	distributed total viscoplastic strains
`flow`	viscoplastic flow
`fun`	element shape functions
`funf`	intermediate array

`funny`	intermediate array
`funnyf`	intermediate array
`gamma`	intermediate array
`gg`	intermediate array
`globma_pp`	distributed global mass matrix
`globma_tmp`	temporary storage mass mass vector
`jac`	Jacobian matrix
`kay`	conductivity property matrix
`kc`	conductivity matrix
`kcx`	x contribution to conductivity matrix
`kcy`	y contribution to conductivity matrix
`kcz`	z contribution to conductivity matrix
`kd`	total element matrix
`ke`	element "stiffness" matrix
`km`	element stiffness matrix
`loads_pp`	distributed loads vector
`mass`	element lumped mass matrix
`mm_pp`	distributed mass vector
`mm_tmp`	temporary vector
`m1`	plastic potential derivatives matrix
`m2`	plastic potential derivatives matrix
`m3`	plastic potential derivatives matrix
`newlo_pp`	distributed new loads
`oldis_pp`	previous distributed displacements
`pl`	plasticity matrix
`pm`	element mass matrix
`points`	integrating point local coordinates
`p_g_co_pp`	distributed nodal coordinates
`pmul_pp`	gather-scatter matrix
`p_pp`	distributed pcg vector
`qinc`	vector of load increment terms
`row`	row array
`rowf`	row of fluid derivative matrix
`row1`	
`row2`	intermediate arrays
`row3`	
`rt_pp`	distributed vector
`r_pp`	distributed pcg vector
`s`	intermediate array
`sigma`	element stresses
`store_mm_pp`	distributed element mass storage
`store_pm_pp`	distributed pm matrices
`store_pp`	distributed penalty storage
`storka_pp`	distributed storage of pm and km
`storkb_pp`	distributed storage of pm and km
`storkd_pp`	distributed `kd` matrices
`storke_pp`	distributed `ke` matrices

`storkm_pp`	distributed stored element stiffness matrices
`stress`	stress vector
`temp_pp`	distributed intermediate vector
`tensor_pp`	distributed stresses
`totd_pp`	distributed total displacements
`ua_pp`	distributed $\{\mathbf{U}\}$ in product $\{\mathbf{U}\} + [\mathbf{A}]\{\mathbf{V}\}$
`utemp_pp`	gather-scatter matrix
`uvel`	x-velocity
`u_pp`	distributed pcg vector
`val`	prescribed load/displacement values
`vvel`	y-velocity
`val_f`	values of fixed freedoms
`va_pp`	distributed $\{\mathbf{V}\}$ in product $\{\mathbf{U}\} + [\mathbf{A}]\{\mathbf{V}\}$
`vdiag_pp`	distributed diagonal vector
`vol`	array for volumetric strain
`volf`	array for fluid volumetric strain
`vu_pp`	distributed intermediate vector
`v_store_pp`	distributed stored Lanczos vectors
`weights`	weighting coefficients
`wvel`	z-velocity
`w1_pp`	intermediate array
`x`	local array
`xnew_pp`	distributed pcg vector
`x0_pp`	distributed old displacement vector
`x1_pp`	distributed new displacement vector
`x_old_pp`	distributed previous x vector
`x_pp`	distributed pcg vector
`y_pp`	distributed y vector
`y1_pp`	distributed y1 vector
`z_pp`	intermediate array

References

HSL 2002 *A Collection of Fortran Codes for Large-scale Scientific Computation*. See `http://www.cse.clrc.ac.uk/nag/hsl/`.

Margetts L 2002 *Parallel Finite Element Analysis*. PhD thesis, University of Manchester, U.K.

Pettipher MA and Smith IM 1997 The development of an MPP implementation of a suite of finite element codes. *High-Performance Computing and Networking: Lecture Notes in Computer Science*. Springer-Verlag, Berlin, pp. 1225:400–409.

Smith IM 2000 A general purpose system for finite element analyses in parallel. *Eng Comput* **17**(1), 75–91.

Smith IM and Margetts L 2003 Portable parallel processing for nonlinear problems. *Proc COMPLAS 2003*, Barcelona.

Smith IM and Wang A 1998 Analysis of piled rafts. *Int J Numer Anal Methods Geomech* **22**(10), 777–790.

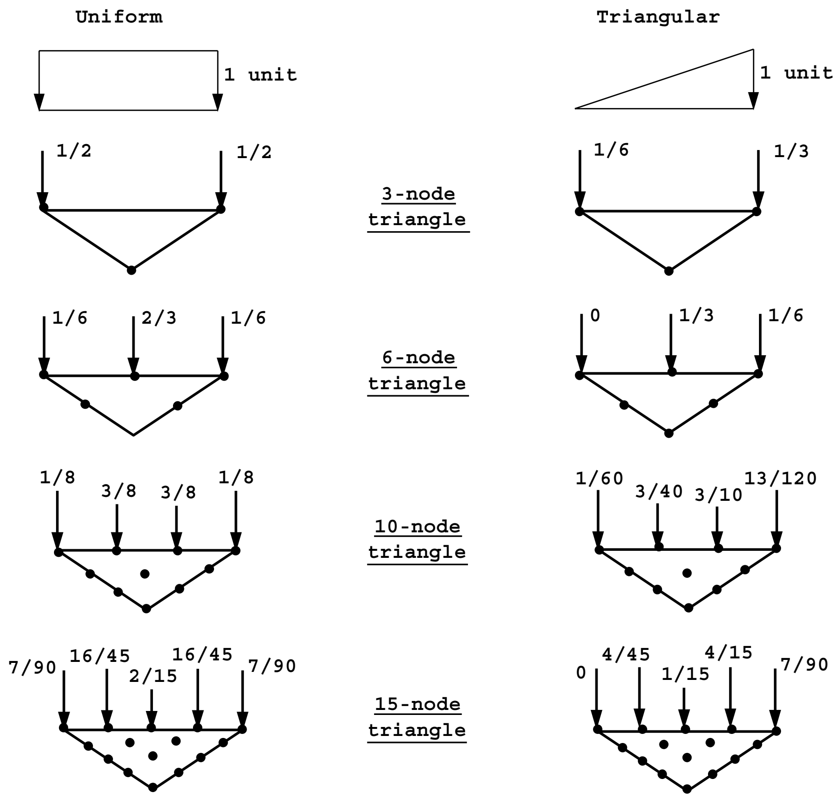
A

Equivalent Nodal Loads

PLANAR ELEMENTS (2D)

Width of loaded face = 1 unit

LOAD TYPE

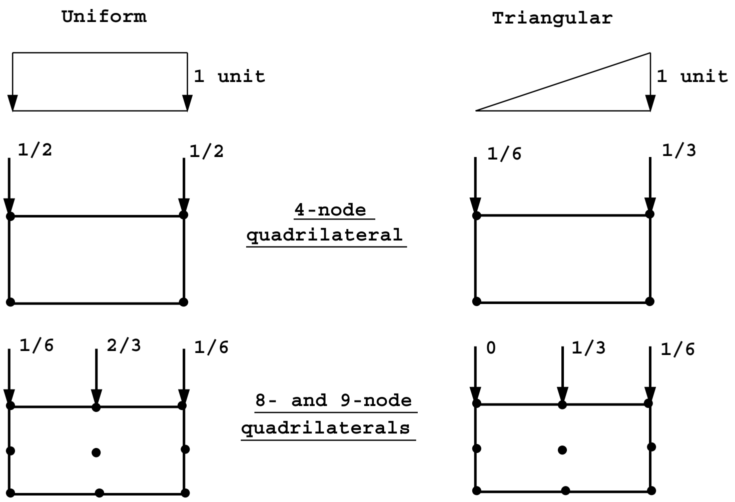


Programming the Finite Element Method 4e I. M. Smith and D. V. Griffiths
© 2004 John Wiley & Sons, Ltd ISBNs: 0-470-84970-3 (PBK); 0-470-84969-X (PPC)

PLANAR ELEMENTS (2D)

Width of loaded face = 1 unit

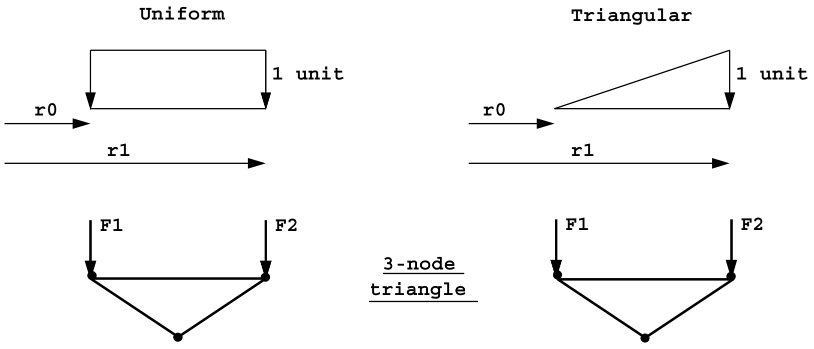
LOAD TYPE



AXISYMMETRIC ELEMENTS (2D)

Loading over 1 radian

LOAD TYPE



$$\mathbf{F}_1 = \tfrac{1}{6}(r_1^2 + r_o r_1 - 2r_o^2) \qquad \mathbf{F_1} = \tfrac{1}{12}(r_1^2 - r_o^2)$$

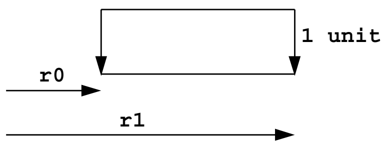
$$\mathbf{F}_2 = \tfrac{1}{6}(2r_1^2 - r_o r_1 - r_o^2) \qquad \mathbf{F_2} = \tfrac{1}{12}(3r_1^2 - 2r_o r_1 - r_o^2)$$

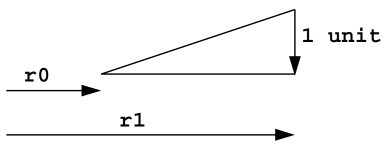
AXISYMMETRIC ELEMENTS (2D)

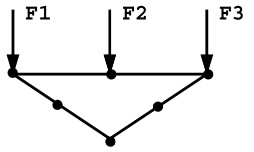
Loading over 1 radian

LOAD TYPE

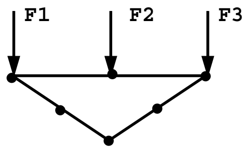
Uniform **Triangular**







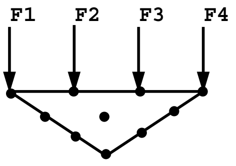
6-node triangle



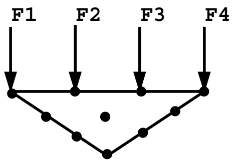
$$\mathbf{F}_1 = \tfrac{1}{6}(r_o r_1 - r_o^2) \qquad \mathbf{F}_1 = -\tfrac{1}{60}(r_1^2 - 2r_o r_1 + r_o^2)$$

$$\mathbf{F}_2 = \tfrac{1}{3}(r_1^2 - r_o^2) \qquad \mathbf{F}_2 = \tfrac{1}{15}(3r_1^2 - r_o r_1 - 2r_o^2)$$

$$\mathbf{F}_3 = \tfrac{1}{6}(r_1^2 - r_o r_1) \qquad \mathbf{F}_3 = \tfrac{1}{60}(9r_1^2 - 8r_o r_1 - r_o^2)$$



10-node triangle



$$\mathbf{F}_1 = \tfrac{1}{120}(2r_1^2 + 11r_o r_1 - 13r_o^2) \qquad \mathbf{F}_1 = \tfrac{1}{120}(r_1^2 - r_o^2)$$

$$\mathbf{F}_2 = \tfrac{1}{40}(3r_1^2 + 9r_o r_1 - 12r_o^2) \qquad \mathbf{F}_2 = \tfrac{3}{40}(r_o r_1 - r_o^2)$$

$$\mathbf{F}_3 = \tfrac{1}{40}(12r_1^2 - 9r_o r_1 - 3r_o^2) \qquad \mathbf{F}_3 = \tfrac{3}{40}(3r_1^2 - 2r_o r_1 - r_o^2)$$

$$\mathbf{F}_4 = \tfrac{1}{120}(13r_1^2 - 11r_o r_1 - 2r_o^2) \qquad \mathbf{F}_4 = \tfrac{1}{120}(12r_1^2 - 11r_o r_1 - r_o^2)$$

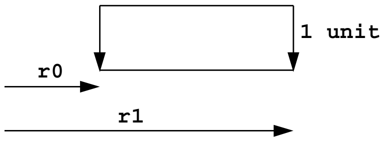
AXISYMMETRIC ELEMENTS (2D)

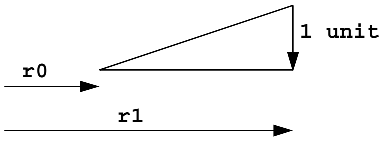
Loading over 1 radian

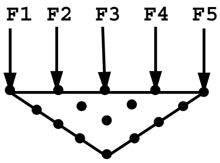
LOAD TYPE

Uniform

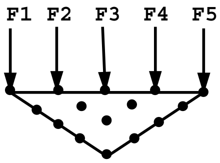
Triangular







15-node triangle



$\mathbf{F}_1 = \frac{7}{90}(r_o r_1 - r_o^2)$

$\mathbf{F}_2 = \frac{4}{45}(r_1^2 + 2r_o r_1 - 3r_o^2)$

$\mathbf{F}_3 = \frac{1}{15}(r_1^2 - r_o^2)$

$\mathbf{F}_4 = \frac{4}{45}(3r_1^2 - 2r_o r_1 - r_o^2)$

$\mathbf{F}_5 = \frac{7}{90}(r_1^2 - r_o r_1)$

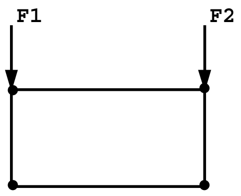
$\mathbf{F}_1 = -\frac{1}{252}(r_1^2 - 2r_o r_1 + r_o^2)$

$\mathbf{F}_2 = \frac{4}{315}(3r_1^2 + r_o r_1 - 4r_o^2)$

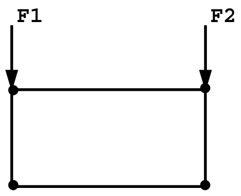
$\mathbf{F}_3 = \frac{1}{105}(r_1^2 + 5r_o r_1 - 6r_o^2)$

$\mathbf{F}_4 = \frac{4}{315}(17r_1^2 - 13r_o r_1 - 4r_o^2)$

$\mathbf{F}_5 = \frac{1}{1260}(93r_1^2 - 88r_o r_1 - 5r_o^2)$



4-node quadrilateral

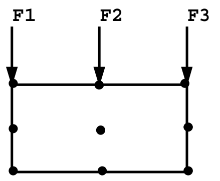


$\mathbf{F}_1 = \frac{1}{6}(r_1^2 + r_o r_1 - 2r_o^2)$

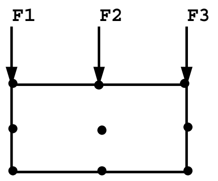
$\mathbf{F}_2 = \frac{1}{6}(2r_1^2 - r_o r_1 - r_o^2)$

$\mathbf{F}_1 = \frac{1}{12}(r_1^2 - r_o^2)$

$\mathbf{F}_2 = \frac{1}{12}(3r_1^2 - 2r_o r_1 - r_o^2)$



8- and 9-node quadrilaterals



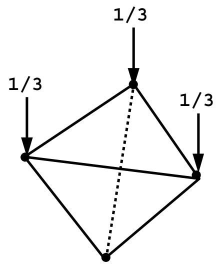
$$\mathbf{F}_1 = \tfrac{1}{6}(r_o r_1 - r_o^2)$$
$$\mathbf{F}_2 = \tfrac{1}{3}(r_1^2 - r_o^2)$$
$$\mathbf{F}_3 = \tfrac{1}{6}(r_1^2 - r_o r_1)$$

$$\mathbf{F}_1 = -\tfrac{1}{60}(r_1^2 - 2r_o r_1 + r_o^2)$$
$$\mathbf{F}_2 = \tfrac{1}{15}(3r_1^2 - r_o r_1 - 2r_o^2)$$
$$\mathbf{F}_3 = \tfrac{1}{60}(9r_1^2 - 8r_o r_1 - r_o^2)$$

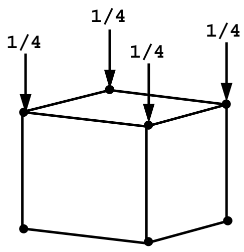
THREE DIMENSIONAL ELEMENTS (3D)

Area of loaded face = 1 unit
Unit stress applied

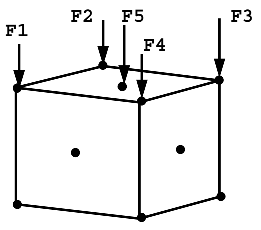
4-node tetrahedron



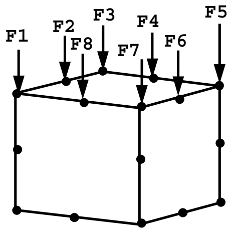
8-node hexahedron



14-node hexahedron (type 6)



20-node hexahedron



$$\mathbf{F}_1 = \mathbf{F}_2 = \mathbf{F}_3 = \mathbf{F}_4 = \tfrac{1}{12}$$
$$\mathbf{F}_5 = \tfrac{2}{3}$$

$$\mathbf{F}_1 = \mathbf{F}_3 = \mathbf{F}_5 = \mathbf{F}_7 = -\tfrac{1}{12}$$
$$\mathbf{F}_2 = \mathbf{F}_4 = \mathbf{F}_6 = \mathbf{F}_8 = \tfrac{1}{3}$$

B

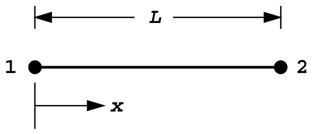
Shape Functions and Element Node Numbering

1D elements

2-node rod

$$N_1 = 1 - \tfrac{x}{L}$$

$$N_2 = \tfrac{x}{L}$$



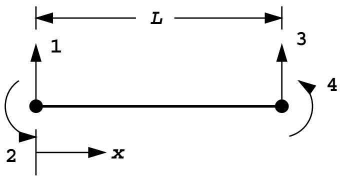
2-node beam

$$N_1 = \tfrac{1}{L^3}(L^3 - 3Lx^2 + 2x^3)$$

$$N_2 = \tfrac{1}{L^2}(L^2x - 2Lx^2 + x^3)$$

$$N_3 = \tfrac{1}{L^3}(3Lx^2 - 2x^3)$$

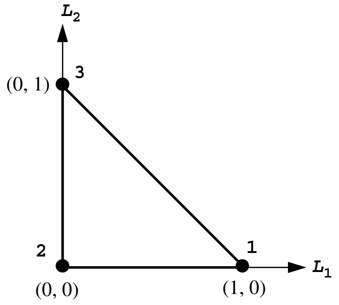
$$N_4 = \tfrac{1}{L^2}(x^3 - Lx^2)$$



2D elements

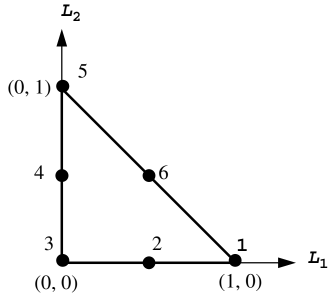
3-node triangle

$$N_1 = L_1$$
$$N_2 = (1 - L_1 - L_2)$$
$$N_3 = L_2$$



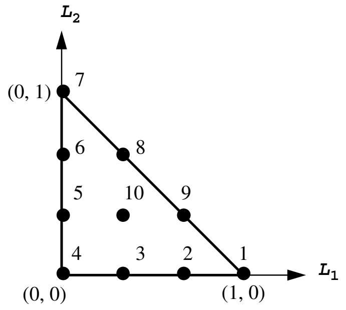
6-node triangle

$$N_1 = (2L_1 - 1)L_1$$
$$N_2 = 4(1 - L_1 - L_2)L_1$$
$$N_3 = (2(1 - L_1 - L_2) - 1)(1 - L_1 - L_2)$$
$$N_4 = 4L_2(1 - L_1 - L_2)$$
$$N_5 = (2L_2 - 1)L_2$$
$$N_6 = 4L_1L_2$$

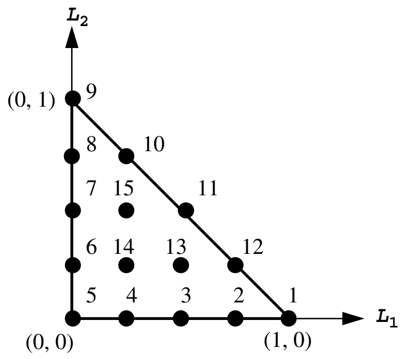


10-node triangle

$$N_1 = \tfrac{1}{2}(3L_1 - 1)(3L_1 - 2)L_1$$
$$N_2 = -\tfrac{9}{2}(L_2 + L_1 - 1)(3L_1 - 1)L_1$$
$$N_3 = \tfrac{9}{2}(3L_2 + 3L_1 - 2)(L_2 + L_1 - 1)L_1$$
$$N_4 = -\tfrac{1}{2}(3L_2 + 3L_1 - 1)(3L_2 + 3L_1 - 2)(L_2 + L_1 - 1)$$
$$N_5 = \tfrac{9}{2}(3L_2 + 3L_1 - 2)(L_2 + L_1 - 1)L_2$$
$$N_6 = -\tfrac{9}{2}(3L_2 - 1)(L_2 + L_1 - 1)L_2$$
$$N_7 = \tfrac{1}{2}(3L_2 - 1)(3L_2 - 2)L_2$$
$$N_8 = \tfrac{9}{2}(3L_2 - 1)L_2L_1$$
$$N_9 = \tfrac{9}{2}(3L_1 - 1)L_2L_1$$
$$N_{10} = -27(L_2 + L_1 - 1)L_2L_1$$



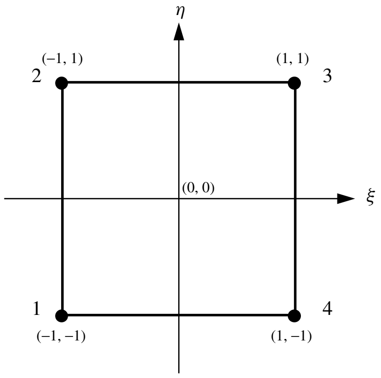
15-node triangle



$$
\begin{aligned}
N_1 &= \tfrac{32}{3}L_1(L_1-\tfrac{1}{4})(L_1-\tfrac{1}{2})(L_1-\tfrac{3}{4})\\
N_2 &= \tfrac{128}{3}L_1(1-L_1-L_2)(L_1-\tfrac{1}{4})(L_1-\tfrac{1}{2})\\
N_3 &= 64L_1(1-L_1-L_2)(L_1-\tfrac{1}{4})(1-L_2-L_1-\tfrac{1}{4})\\
N_4 &= \tfrac{128}{3}L_1(1-L_1-L_2)(1-L_2-L_1-\tfrac{1}{4})(1-L_2-L_1-\tfrac{1}{2})\\
N_5 &= \tfrac{32}{3}(1-L_1-L_2)(1-L_2-L_1-\tfrac{1}{4})(1-L_2-L_1-\tfrac{1}{2})(1-L_2-L_1-\tfrac{3}{4})\\
N_6 &= \tfrac{128}{3}(1-L_1-L_2)L_2(1-L_2-L_1-\tfrac{1}{4})(1-L_2-L_1-\tfrac{1}{2})\\
N_7 &= 64(1-L_1-L_2)L_2(L_2-\tfrac{1}{4})(1-L_1-L_2-\tfrac{1}{4})\\
N_8 &= \tfrac{128}{3}(1-L_1-L_2)L_2(L_2-\tfrac{1}{4})(L_2-\tfrac{1}{2})\\
N_9 &= \tfrac{32}{3}L_2(L_2-\tfrac{1}{4})(L_2-\tfrac{1}{2})(L_2-\tfrac{3}{4})\\
N_{10} &= \tfrac{128}{3}L_2L_1(L_2-\tfrac{1}{4})(L_2-\tfrac{1}{2})\\
N_{11} &= 64L_2L_1(L_2-\tfrac{1}{4})(L_1-\tfrac{1}{4})\\
N_{12} &= \tfrac{128}{3}L_2L_1(L_1-\tfrac{1}{4})(L_1-\tfrac{1}{2})\\
N_{13} &= 128L_2L_1(1-L_1-L_2)(L_1-\tfrac{1}{4})\\
N_{14} &= 128L_2L_1(1-L_1-L_2)(1-L_2-L_1-\tfrac{1}{4})\\
N_{15} &= 128L_2L_1(L_2-\tfrac{1}{4})(1-L_1-L_2)
\end{aligned}
$$

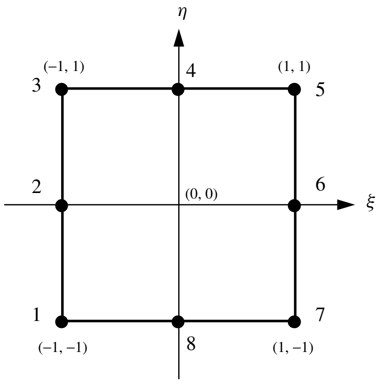
4-node quadrilateral

$$\begin{aligned}
N_1 &= \tfrac{1}{4}(1-\xi)(1-\eta)\\
N_2 &= \tfrac{1}{4}(1-\xi)(1+\eta)\\
N_3 &= \tfrac{1}{4}(1+\xi)(1+\eta)\\
N_4 &= \tfrac{1}{4}(1+\xi)(1-\eta)
\end{aligned}$$



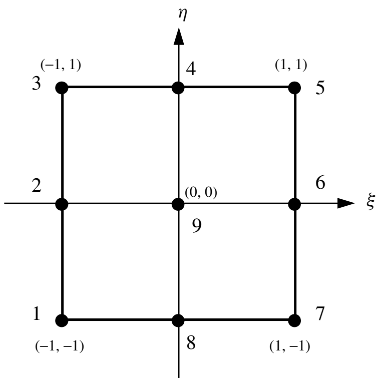
8-node quadrilateral

$$\begin{aligned}
N_1 &= \tfrac{1}{4}(1-\xi)(1-\eta)(-\xi-\eta-1)\\
N_2 &= \tfrac{1}{2}(1-\xi)(1-\eta^2)\\
N_3 &= \tfrac{1}{4}(1-\xi)(1+\eta)(-\xi+\eta-1)\\
N_4 &= \tfrac{1}{2}(1-\xi^2)(1+\eta)\\
N_5 &= \tfrac{1}{4}(1+\xi)(1+\eta)(\xi+\eta-1)\\
N_6 &= \tfrac{1}{2}(1+\xi)(1-\eta^2)\\
N_7 &= \tfrac{1}{4}(1+\xi)(1-\eta)(\xi-\eta-1)\\
N_8 &= \tfrac{1}{2}(1-\xi^2)(1-\eta)
\end{aligned}$$



9-node quadrilateral

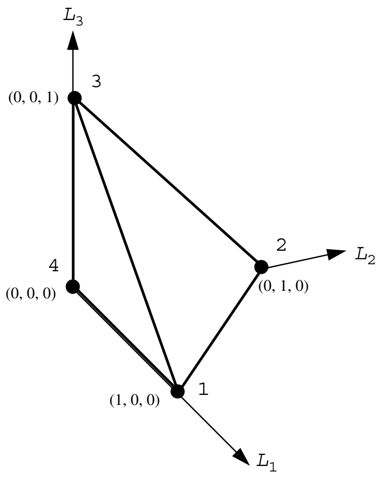
$$\begin{aligned}
N_1 &= \tfrac{1}{4}\xi(\xi-1)\eta(\eta-1)\\
N_2 &= -\tfrac{1}{2}\xi(\xi-1)(\eta+1)(\eta-1)\\
N_3 &= \tfrac{1}{4}\xi(\xi-1)\eta(\eta+1)\\
N_4 &= -\tfrac{1}{2}(\xi+1)(\xi-1)\eta(\eta+1)\\
N_5 &= \tfrac{1}{4}\xi(\xi+1)\eta(\eta+1)\\
N_6 &= -\tfrac{1}{2}\xi(\xi+1)(\eta+1)(\eta-1)\\
N_7 &= \tfrac{1}{4}\xi(\xi+1)\eta(\eta-1)\\
N_8 &= -\tfrac{1}{2}(\xi+1)(\xi-1)\eta(\eta-1)\\
N_9 &= (\xi+1)(\xi-1)(\eta+1)(\eta-1)
\end{aligned}$$



3D elements

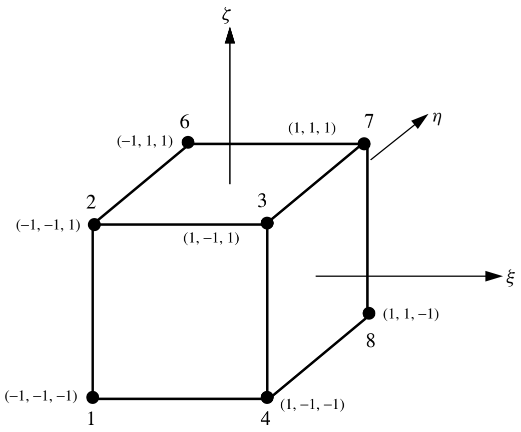
4-node tetrahedron

$$\begin{aligned} N_1 &= L_1 \\ N_2 &= L_2 \\ N_3 &= L_3 \\ N_4 &= (1 - L_1 - L_2 - L_3) \end{aligned}$$

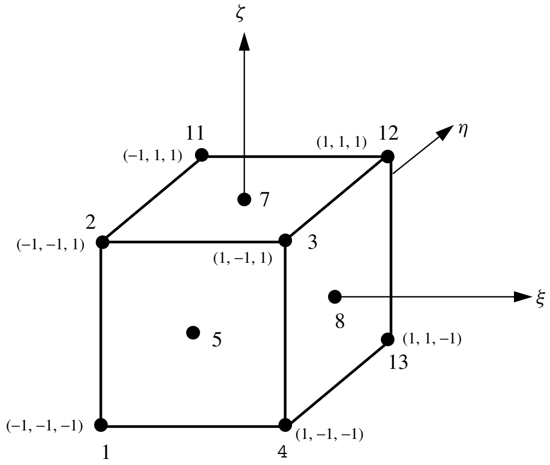


8-node hexahedron

$$\begin{aligned} N_1 &= \tfrac{1}{8}(1-\xi)(1-\eta)(1-\zeta) \\ N_2 &= \tfrac{1}{8}(1-\xi)(1-\eta)(1+\zeta) \\ N_3 &= \tfrac{1}{8}(1+\xi)(1-\eta)(1+\zeta) \\ N_4 &= \tfrac{1}{8}(1+\xi)(1-\eta)(1-\zeta) \\ N_5 &= \tfrac{1}{8}(1-\xi)(1+\eta)(1-\zeta) \\ N_6 &= \tfrac{1}{8}(1-\xi)(1+\eta)(1+\zeta) \\ N_7 &= \tfrac{1}{8}(1+\xi)(1+\eta)(1+\zeta) \\ N_8 &= \tfrac{1}{8}(1+\xi)(1+\eta)(1-\zeta) \end{aligned}$$

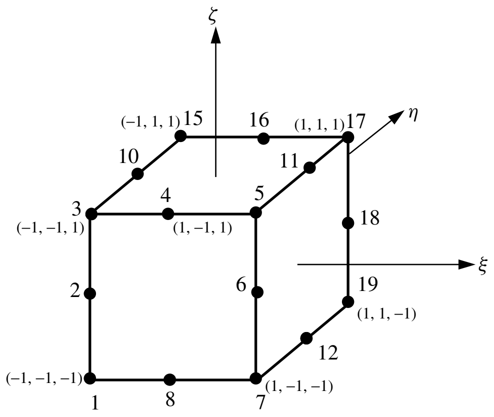


14-node hexahedron (Type 6)



$$
\begin{aligned}
N_1 &= \tfrac{1}{8}(\xi\eta + \xi\zeta + 2\xi + \eta\zeta + 2\eta + 2\zeta + 2)(\xi - 1)(\eta - 1)(\zeta - 1) \\
N_2 &= -\tfrac{1}{8}(\xi\eta - \xi\zeta + 2\xi - \eta\zeta + 2\eta - 2\zeta + 2)(\xi - 1)(\eta - 1)(\zeta + 1) \\
N_3 &= -\tfrac{1}{8}(\xi\eta - \xi\zeta + 2\xi + \eta\zeta - 2\eta + 2\zeta - 2)(\xi + 1)(\eta - 1)(\zeta + 1) \\
N_4 &= \tfrac{1}{8}(\xi\eta + \xi\zeta + 2\xi - \eta\zeta - 2\eta - 2\zeta - 2)(\xi + 1)(\eta - 1)(\zeta - 1) \\
N_5 &= -\tfrac{1}{2}(\xi + 1)(\xi - 1)(\eta - 1)(\zeta + 1)(\zeta - 1) \\
N_6 &= -\tfrac{1}{2}(\xi - 1)(\eta + 1)(\eta - 1)(\zeta + 1)(\zeta - 1) \\
N_7 &= \tfrac{1}{2}(\xi + 1)(\xi - 1)(\eta + 1)(\eta - 1)(\zeta + 1) \\
N_8 &= \tfrac{1}{2}(\xi + 1)(\eta + 1)(\eta - 1)(\zeta + 1)(\zeta - 1) \\
N_9 &= -\tfrac{1}{2}(\xi + 1)(\xi - 1)(\eta + 1)(\eta - 1)(\zeta - 1) \\
N_{10} &= \tfrac{1}{8}(\xi\eta - \xi\zeta - 2\xi + \eta\zeta + 2\eta - 2\zeta - 2)(\xi - 1)(\eta + 1)(\zeta - 1) \\
N_{11} &= -\tfrac{1}{8}(\xi\eta + \xi\zeta - 2\xi - \eta\zeta + 2\eta + 2\zeta - 2)(\xi - 1)(\eta + 1)(\zeta + 1) \\
N_{12} &= -\tfrac{1}{8}(\xi\eta + \xi\zeta - 2\xi + \eta\zeta - 2\eta - 2\zeta + 2)(\xi + 1)(\eta + 1)(\zeta + 1) \\
N_{13} &= \tfrac{1}{8}(\xi\eta - \xi\zeta - 2\xi - \eta\zeta - 2\eta + 2\zeta + 2)(\xi + 1)(\eta + 1)(\zeta - 1) \\
N_{14} &= \tfrac{1}{2}(\xi + 1)(\xi - 1)(\eta + 1)(\zeta + 1)(\zeta - 1)
\end{aligned}
$$

20-node hexahedron



$$
\begin{aligned}
N_1 &= \tfrac{1}{8}(1-\xi)(1-\eta)(1-\zeta)(-\xi-\eta-\zeta-2) \\
N_2 &= \tfrac{1}{4}(1-\xi)(1-\eta)(1-\zeta^2) \\
N_3 &= \tfrac{1}{8}(1-\xi)(1-\eta)(1+\zeta)(-\xi-\eta+\zeta-2) \\
N_4 &= \tfrac{1}{4}(1-\xi^2)(1-\eta)(1+\zeta) \\
N_5 &= \tfrac{1}{8}(1+\xi)(1-\eta)(1+\zeta)(\xi-\eta+\zeta-2) \\
N_6 &= \tfrac{1}{4}(1+\xi)(1-\eta)(1-\zeta^2) \\
N_7 &= \tfrac{1}{8}(1+\xi)(1-\eta)(1-\zeta)(\xi-\eta-\zeta-2) \\
N_8 &= \tfrac{1}{4}(1-\xi^2)(1-\eta)(1-\zeta) \\
N_9 &= \tfrac{1}{4}(1-\xi)(1-\eta^2)(1-\zeta) \\
N_{10} &= \tfrac{1}{4}(1-\xi)(1-\eta^2)(1+\zeta) \\
N_{11} &= \tfrac{1}{4}(1+\xi)(1-\eta^2)(1+\zeta) \\
N_{12} &= \tfrac{1}{4}(1+\xi)(1-\eta^2)(1-\zeta) \\
N_{13} &= \tfrac{1}{8}(1-\xi)(1+\eta)(1-\zeta)(-\xi+\eta-\zeta-2) \\
N_{14} &= \tfrac{1}{4}(1-\xi)(1+\eta)(1-\zeta^2) \\
N_{15} &= \tfrac{1}{8}(1-\xi)(1+\eta)(1+\zeta)(-\xi+\eta+\zeta-2) \\
N_{16} &= \tfrac{1}{4}(1-\xi^2)(1+\eta)(1+\zeta) \\
N_{17} &= \tfrac{1}{8}(1+\xi)(1+\eta)(1+\zeta)(\xi+\eta+\zeta-2) \\
N_{18} &= \tfrac{1}{4}(1+\xi)(1+\eta)(1-\zeta^2) \\
N_{19} &= \tfrac{1}{8}(1+\xi)(1+\eta)(1-\zeta)(\xi+\eta-\zeta-2) \\
N_{20} &= \tfrac{1}{4}(1-\xi^2)(1+\eta)(1-\zeta)
\end{aligned}
$$

C

Plastic Stress–strain Matrices and Plastic Potential Derivatives

The following expressions give the plastic stress–strain matrices for 2D applications using von Mises (Yamada *et al.* 1968) and Mohr–Coulomb (Griffiths and Willson 1986, see Chapter 6 references). For 3D applications, the expressions are more conveniently generated using computer algebra systems (see subroutines `vmdpl` and `mcdpl` for von Mises and Mohr–Coulomb respectively).

A. PLASTIC STRESS–STRAIN MATRICES

1. VON MISES

$$[\mathbf{D}^p] = \frac{2G}{t^2}\begin{bmatrix} s_x^2 & s_x s_y & s_x \tau_{xy} & s_z s_x \\ & s_y^2 & s_y \tau_{xy} & s_y s_z \\ & & \tau_{xy}^2 & s_z \tau_{xy} \\ & \text{symmetrical} & & s_z^2 \end{bmatrix}$$

where

$$G = \text{shear modulus}$$

$$t = \text{second deviatoric stress invariant (6.3)}$$

$$s_x = (2\sigma_x - \sigma_y - \sigma_z)/3, \text{ etc.}$$

1. MOHR–COULOMB

If not near a corner, that is $|\sin\theta| \le 0.49$, then

$$[\mathbf{D}^p] = \frac{E}{2(1+\nu)(1-2\nu)(1-2\nu+\sin\phi\sin\psi)}[\mathbf{A}]$$

where

$$[\mathbf{A}] = \begin{bmatrix} R_1C_1 & R_1C_2 & R_1C_3 & R_1C_4 \\ R_2C_1 & R_2C_2 & R_2C_3 & R_2C_4, \\ R_3C_1 & R_3C_2 & R_3C_3 & R_3C_4 \\ R_4C_1 & R_4C_2 & R_4C_3 & R_4C_4 \end{bmatrix}$$

$$\begin{aligned} C_1 &= \sin\phi + k_1(1-2\nu)\sin\alpha \\ C_2 &= \sin\phi - k_1(1-2\nu)\sin\alpha \\ C_3 &= k_2(1-2\nu)\cos\alpha \\ C_4 &= 2\nu\sin\phi \end{aligned}$$

$$\begin{aligned} R_1 &= \sin\psi + k_1(1-2\nu)\sin\alpha \\ R_2 &= \sin\psi - k_1(1-2\nu)\sin\alpha \\ R_3 &= k_2(1-2\nu)\cos\alpha \\ R_4 &= 2\nu\sin\psi, \end{aligned} \tag{C.1}$$

$$\alpha = \arctan\left|\frac{\sigma_x - \sigma_y}{2\tau_{xy}}\right|$$

and

$$k_1 = \begin{cases} 1 & \text{if } |\sigma_y| \geq |\sigma_x| \\ -1 & \text{if } |\sigma_y| < |\sigma_x| \end{cases}$$

$$k_2 = \begin{cases} 1 & \text{if } \tau_{xy} \geq 0 \\ -1 & \text{if } \tau_{xy} < 0 \end{cases}$$

If near a corner, that is $|\sin\theta| > 0.49$, then

$$[\mathbf{D}^p] = \frac{E}{(1+\nu)(1-2\nu)(K_\phi \sin\psi + C_\phi C_\psi t^2(1-2\nu))}[\mathbf{A}]$$

where $[\mathbf{A}]$ is defined as before with

$$\begin{aligned} C_1 &= K_\phi + C_\phi((1-\nu)s_x + \nu(s_y + s_z)) \\ C_2 &= K_\phi + C_\phi((1-\nu)s_y + \nu(s_z + s_x)) \\ C_3 &= C_\phi(1-2\nu)\tau_{xy} \\ C_4 &= K_\phi + C_\phi((1-\nu)s_z + \nu(s_x + s_y)) \end{aligned} \tag{C.2}$$

$$\begin{aligned} R_1 &= K_\psi + C_\psi((1-\nu)s_x + \nu(s_y + s_z)) \\ R_2 &= K_\psi + C_\psi((1-\nu)s_y + \nu(s_z + s_x)) \end{aligned}$$

$$R_3 = C_\psi(1-2\nu)\tau_{xy}$$
$$R_4 = K_\psi + C_\psi((1-\nu)s_z + \nu(s_x + s_y))$$

and

$$K_\phi = \frac{\sin\phi}{3}(1+\nu)$$
$$K_\psi = \frac{\sin\psi}{3}(1+\nu)$$
$$C_\phi = \frac{\sqrt{6}}{4t}\left(1 \pm \frac{\sin\phi}{3}\right)$$
$$C_\psi = \frac{\sqrt{6}}{4t}\left(1 \pm \frac{\sin\psi}{3}\right)$$

In the expressions for C_ϕ and C_ψ, the positive sign is valid for $\theta \approx -30^\circ$ and the negative sign is valid if $\theta \approx 30^\circ$.

B. PLASTIC POTENTIAL DERIVATIVES

$$\left\{\frac{\partial Q}{\partial \boldsymbol{\sigma}}\right\} = \frac{\partial Q}{\partial \sigma_m}\left\{\frac{\partial \sigma_m}{\partial \boldsymbol{\sigma}}\right\} + \frac{\partial Q}{\partial J_2}\left\{\frac{\partial J_2}{\partial \boldsymbol{\sigma}}\right\} + \frac{\partial Q}{\partial J_3}\left\{\frac{\partial J_3}{\partial \boldsymbol{\sigma}}\right\}$$
$$= \left(\frac{\partial Q}{\partial \sigma_m}[\mathbf{M}^1] + \frac{\partial Q}{\partial J_2}[\mathbf{M}^2] + \frac{\partial Q}{\partial J_3}[\mathbf{M}^3]\right)\{\boldsymbol{\sigma}\}$$

where

$$[\mathbf{M}^1] = \frac{1}{3(\sigma_x + \sigma_y + \sigma_z)}\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ & 1 & 1 & 0 & 0 & 0 \\ & & 1 & 0 & 0 & 0 \\ & & & 0 & 0 & 0 \\ & & & & 0 & 0 \\ \text{symmetrical} & & & & & 0 \end{bmatrix}$$

$$[\mathbf{M}^2] = \frac{1}{3}\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ & 2 & -1 & 0 & 0 & 0 \\ & & 2 & 0 & 0 & 0 \\ & & & 6 & 0 & 0 \\ & & & & 6 & 0 \\ \text{symmetrical} & & & & & 6 \end{bmatrix}$$

$$[\mathbf{M}^3] = \frac{1}{3}\begin{bmatrix} s_x & s_z & s_y & \tau_{xy} & -2\tau_{yz} & \tau_{zx} \\ & s_y & s_x & \tau_{xy} & \tau_{yz} & -2\tau_{zx} \\ & & s_z & -2\tau_{xy} & \tau_{yz} & \tau_{zx} \\ & & & -3s_z & 3\tau_{zx} & 3\tau_{yz} \\ & & & & -3s_x & 3\tau_{xy} \\ & & \text{symmetrical} & & & -3s_y \end{bmatrix}$$

and

$$\{\sigma\} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{Bmatrix}$$

1. VON MISES

$$\frac{\partial Q}{\partial \sigma_m} = 0$$

$$\frac{\partial Q}{\partial J_2} = \sqrt{\frac{3}{2}}\frac{1}{t}$$

$$\frac{\partial Q}{\partial J_3} = 0$$

2. MOHR–COULOMB

$$\frac{\partial Q}{\partial \sigma_m} = \sin\psi$$

$$\frac{\partial Q}{\partial J_2} = \frac{\cos\theta}{\sqrt{2}t}\left(1 + \tan\theta\tan 3\theta + \frac{\sin\psi}{\sqrt{3}}(\tan 3\theta - \tan\theta)\right)$$

$$\frac{\partial Q}{\partial J_3} = \frac{\sqrt{3}\sin\theta + \sin\psi\cos\theta}{t^2\cos 3\theta}$$

D

`main` Library Subroutines

This Appendix describes the "black box" and general purpose subroutines and functions held in a library called `main`, which is attached to the main programs described in Chapters 4 to 11 in this book through the command `USE main`. Subroutines in parentheses are normally called in conjunction with the ones they follow.

All the subroutines except `lancz1` and `lancz2` are listed in full on the web site `www.mines.edu/fs_home/vgriffit/4th_ed/source/library/main`

Subroutine descriptions

The following descriptions indicate the `main` library subroutines and functions in alphabetic order, together with the meaning of their arguments. Arguments in **bold** are those returned by the subroutine.

Name	Arguments	Description
`bandred` `(bisect)`	`a`, **d**, **e**	Returns the transformed diagonal **d** and off-diagonal **e** following transformation of real symmetric band matrix `a` to tridiagonal form by Jacobi rotations.
`bandwidth`	`g`	Function returns the maximum bandwidth for an element with steering vector `g`.
`banmul`	`kb`, `loads`, **ans**	Returns the product of symmetric band matrix `kb` stored as a rectangle, and vector `loads` to give solution vector **ans**.
`bantmul`	`kb`, `loads`, **ans**	Returns the product of unsymmetric band matrix `kb` stored as a rectangle, and vector `loads` to give solution vector **ans**.

Programming the Finite Element Method 4e I. M. Smith and D. V. Griffiths
© 2004 John Wiley & Sons, Ltd ISBNs: 0-470-84970-3 (PBK); 0-470-84969-X (PPC)

Name	Arguments	Description
beam_ge	**ge**, ell	Returns geometric matrix **ge** for beam element of length ell.
beam_km	**km**, ei, ell	Returns stiffness matrix **km** for beam element of stiffness ei and length ell.
beam_mm	**mm**, fs, ell	Returns "mass" matrix **mm** for beam element of density ρA (or foundation stiffness fs) and length ell.
beemat	**bee**, deriv	Returns **bee** matrix for shape function derivatives deriv.
bee8	**bee**, coord, xi, eta, **det**	Returns "analytical" form of **bee** matrix and Jacobian determinant **det**, for plane 8-node element with nodal coordinates coord at local coordinates xi,eta.
bisect (bandred)	**d**, e, acheps, **ifail**	Returns eigenvalues of a tridiagonal matrix whose leading diagonal is **d** and off-diagonal e. Tolerance is acheps and failure flag **ifail**. Eigenvalues overwrite **d**.
bmat_nonaxi	**bee**, **radius**, coord, deriv, fun, iflag, lth	Returns **bee** matrix for axisymmetric bodies subjected to non-axisymmetric loading from shape functions fun and derivatives deriv. **radius** is the r-coordinate of the Gauss point. lth is the harmonic, and iflag equals 1 for symmetry and -1 for anti-symmetry.
checon	loads, oldlds, tol, **converged**	Returns logical variable **converged** set to .FALSE. if relative change in loads and oldlds is less than tol
contour	loads, g_coord, g_num, ned, ips	Generates a PostScript contour map to output channel ips with file name fe95.con. g_coord and g_num hold nodal coordinates and element connectivity. loads holds the nodal values to be contoured. ned holds number of required contour intervals. 4-node quads only.

Name	Arguments	Description
cross_product	b, c, **a**	Returns matrix **a** which is the cross-product of column b and row c.
deemat	**dee**, e, v	Returns elastic stress–strain **dee** matrix in 2D (plane strain) or 3D. e and v are Young's modulus and Poisson's ratio.
determinant	jac	Function returns the determinant of 2D or 3D square matrix jac
dismsh	loads, nf, ratmax, g_coord, g_num, ips	Generates a PostScript image of the deformed mesh to output channel ips with file name fe95.dis. g_coord and g_num hold nodal coordinates and element connectivity. loads holds the nodal displacements. nf is the nodal freedom array. ratmax holds the ratio of the maximum nodal displacement as plotted as a proportion of the longest *x*- or *y*-dimension of the mesh.
ecmat	**ecm**, fun, ndof, nodof	Returns the consistent mass matrix **ecm** for an element with shape functions fun, ndof freedoms and nodof freedoms per node.
exc_nods	noexe, exele, g_num, **totex**, **ntote**, **nf**	Returns the modified **nf** array accounting for excavated elements. Updates **totex** and **ntote**. noexe holds number of elements to be excavated and exele the element numbers.
fkdiag	**kdiag**, g	Returns the bandwidth vector **kdiag** for the rows of a skyline storage system from g.
fmacat	vmfl, **acat**	Returns an intermediate matrix **acat** from the von Mises flow vector vmfl as used in Programs 6.5 and 6.6.
fmdsig	**dee**, e, v	Returns the elastic stress–strain **dee** matrix in 2D (plane-stress). e and v are Young's modulus and Poisson's ratio.

Name	Arguments	Description
`fmkdke`	`km`, `kp`, `c`, **ke**, **kd**, `theta`	Returns the coupled matrices **ke** and **kd** from the elastic stiffness matrix `km`, conductivity matrix `kp` and coupling matrix `c`. `theta` is the scalar time-stepping parameter.
`fmplat`	**d2x**, **d2y**, **d2xy**, `points`, `aa`, `bb`, `i`	Returns derivative terms **d2x**, **d2y** and **d2xy** for the $\mathtt{i}^{th}$ Gauss point in a rectangular plate element of dimensions `aa` and `bb`. `points` holds the locations of the Gauss points.
`fmrmat`	`vmfl`, `dsbar`, `dlam`, `dee`, **rmat**	Returns matrix **rmat** from the von Mises flow vector `vmfl`, invariant `dsbar`, plastic multplier `dlam` and elastic matrix `dee` as used in equation (6.73).
`formaa`	`vmfl`, `rmat` **daatd**	Returns modified matrix **daatd** from the von Mises flow vector `vmfl`, and matrix `rmat` as used in equation (6.67).
`formke`	`km`, `kp`, `c`, **ke**, `theta`	Returns the coupled matrix **ke** from the elastic stiffness matrix `km`, conductivity matrix `kp` and coupling matrix `c`. `theta` is the scalar time-stepping parameter.
`formku`	**ku**, `km`, `g`	Returns upper triangular global band matrix **ku** stored as a rectangle, from symmetric element matrix `km` and steering vector `g`.
`formlump`	**diag**, `emm`, `g`	Returns lumped global mass matrix as a vector **diag** from consistent element mass matrix `emm` and steering vector `g`.
`formm`	`stress`, **m1**, **m2**, **m3**	Returns matrices **m1**, **m2** and **m3** from stresses `stress` as used in calculation of $\{\partial Q/\partial\sigma\}$ (see equation 6.25).
`formnf`	**nf**	Returns nodal freedom array **nf** from boundary conditions input of 0 s and 1 s.
`formtb`	**pb**, `km`, `g`	Returns global full band matrix **pb** stored as a rectangle from unsymmetric element matrices `km` and steering vectors `g`.

Name	Arguments	Description
`formupv`	**ke**, `c11`, `c12`, `c21`, `c23`, `c32`	Returns unsymmetric element matrix **ke** from constituent matrices `c11`, `c12`, `c21`, `c23` and `c32` for use in *u-p-v* version of Navier–Stokes equations.
`form_s`	`gg`, `ell`, `kappa`, **omega**, **gamma**, **s**	Returns scalar **omega**, and vectors **gamma** and **s** from array `gg`, scalar `kappa` and integer `ell` in BiCGStab.
`fsparv`	**kv**, `km`, `g`, `kdiag`	Returns lower triangular global matrix **kv** stored as a vector in skyline form, from symmetric element matrices `km` and steering vectors `g`. `kdiag` holds the locations of the diagonal terms.
`gauss_band` `(solve_band)`	**pb**, **work**	Returns (Gaussian) factorised unsymmetric full band matrix **pb** and "working" array **work**.
`glob_to_axial`	**axial**, `global`, `coord`	Returns axial force **axial** in 2D or 3D rod element from global force components held in `global`. Nodal coordinates held in `coord`.
`glob_to_loc`	**local**, `global`, `gamma`, `coord`	Returns local components **local** of force and moments from global components held in `global`. `gamma` holds element orientation angle (3D only) and `coord` holds the nodal coordinates. Called by `SUBROUTINE hinge`.
`hinge`	`coord`, `holdr`, `action`, **react**, `prop`, `iel`, `etype`, `gamma`	Returns correction reactions **react** from existing and incremental reactions `holdr` and `action` respectively. `coord` holds nodal coordinates, `prop` holds beam properties, `iel` holds element number, `etype` holds element type, `gamma` holds element orientation angle (3D only).
`invar`	`stress`, **sigm**, **dsbar**, **theta**	Returns stress invariants **sigma**, **dsbar** (equation 6.4) and Lode angle **theta** (equation 6.3), from current stresses held in `stress`.

Name	**Arguments**	**Description**
`interp`	`k`, `dtim`, `rt`, `rl`, **al**, `nstep`	Returns the load/time functions in **al** at the calculation time step resolution by linear interpolation. `k` holds the load/time function number, `dtim` holds the calculation time step, `rt` and `rl` hold the input load/time function and `nstep` holds the required number of calculation time steps.(Program 11.1)
`invert`	**matrix**	Returns the inverse of a small matrix called **matrix** onto itself.
`lancz1` (`lancz2`)	`n`, `el`, `er`, `acc`, `leig`, `lx`, `lalfa`, `lp`, **iflag**, **ua**, **va**, **eig**, **jeig**, **neig**, `x`, `del`, `nu`, **alfa**, **beta**, **v_store**	Lanczos method for eigenvalues See Chapter 10 and reference HSL (2002) for details.
`lancz2` (`lancz1`)	`n`, `lalfa`, `lp`, `eig`, `jeig`, `neig`, `alfa`, `beta`, `lz`, **jflag**, **y**, `w1`, `z`, `v_store`	Lanczos method for eigenvectors See Chapter 10 and reference HSL (2002) for details.
`linmul_sky`	`kv`, `disps`, **loads**, `kdiag`	Returns the product of symmetric matrix `kv` and vector `disps` to give solution vector **loads**. `kv` is stored as a vector in skyline form, `kdiag` holds the diagonal locations in `kv`.
`load_function`	`lf`, `dtim`, **al**	Returns the load/time function in **al** at the calculation time step resolution by linear interpolation. `lf` holds the input load/time function and `dtim` holds the calculation time step. (Chapter 9)
`loc_to_glob`	`local`, **global**, `gamma`, `coord`	Returns global components **global** of force and moments from local components held in `local`. `gamma` holds element orientation angle (3D only) and `coord` holds the nodal coordinates. Called by `SUBROUTINE hinge`.

Name	**Arguments**	**Description**
mesh	g_coord, g_num, ips	Generates a PostScript image of the initial (undeformed) mesh to output channel ips with file name fe95.msh. g_coord and g_num hold nodal coordinates and element connectivity.
mcdpl	phi, psi, dee stress, **pl**	Returns the plastic stress–strain matrix **pl** for a Mohr–Coulomb material from the friction angle phi and dilation angle psi (in degrees). stress holds the stresses and dee holds the elastic stress–strain matrix.
mocouf	phi, c sigm, dsbar, theta, **f**	Returns the Mohr–Coulomb failure function **f**, from the strength parameters phi and c and stress invariants sigm, dsbar and theta.
mocouq	psi, dsbar, theta, **dq1**, **dq2**, **dq3**	Returns the plastic potential terms **dq1**, **dq2** and **dq3** for a Mohr–Coulomb material from dilation angle psi (in degrees) and invariants dsbar and theta.
norm	x	Returns the l2-norm of vector x.
num_to_g	num, nf, **g**	Returns the element steering vector **g** from the element node numbering num and the nodal freedom array nf.
pin_jointed	**km**, ea, coord	Returns the stiffness matrix **km** of a rod element in 2D or 3D. ea holds the element stiffness and coord holds the element nodal coordinates.
rect_km	**km**, coord, e, v	Returns the analytical stiffness matrix **km** of a *rectangular* plane strain 4- or 8-node quadrilateral element assuming 4 Gauss points. coord holds the element nodal coordinates, e and v hold Young's modulus and Poisson's ratio respectively.

Name	Arguments	Description
`rigid_jointed`	**km**, `prop`, `gamma`, `etype`, `iel`, `coord`	Returns the stiffness matrix **km** of a beam–rod element in 2D or 3D. `coord` holds nodal coordinates, `prop` holds beam properties, `iel` holds element number, `etype` holds element type, `gamma` holds element orientation angle (3D only).
`rod_km`	**km**, `ea`, `length`	Returns the stiffness matrix **km** of a rod element in 1D. `ea` holds the element stiffness and `length` holds the element length.
`rod_mm`	**mm**, `length`	Returns the "mass" matrix **mm** of a rod element of unit mass. `length` holds the element length.
`sample`	`element`, **s**, **wt**	Returns the local coordinates **s** and weighting coefficients **wt** for numerical integration of a finite element of type `element`.
`seep4`	**kp**, `coord`, `perm`	Returns the "analytical" conductivity matrix **kp** of a 4-node plane element based on 4 Gauss points. `coord` holds the element nodal coordinates and `perm` holds the permeability matrix.
`shape_der`	**der**, `points`, `i`	Returns the shape function derivatives **der** at the $\texttt{i}^{th}$ integrating point. `points` holds the local coordinates of the integrating points.
`shape_fun`	**fun**, `points`, `i`	Returns the shape functions **fun** at the $\texttt{i}^{th}$ integrating point. `points` holds the local coordinates of the integrating points.

Name	Arguments	Description
solve_band (gauss_band)	pb, work, **loads**	Returns solution **loads** which overwrites RHS by forward and back substitution on (Gaussian) factorised unsymmetric full band matrix pb. work holds "working" array.
spabac (sparin)	kv, **loads**, kdiag	Returns solution **loads** which overwrites RHS by forward and back substitution on (Cholesky) factorised vector kv stored as a skyline. kdiag holds the locations of the diagonal terms.
spabac_gauss (sparin_gauss)	kv, **loads**, kdiag	Returns solution **loads** which overwrites RHS by forward and back substitution on (Gaussian) factorised vector kv stored as a skyline. kdiag holds the locations of the diagonal terms.
sparin (spabac)	**kv**, kdiag	Returns the (Cholesky) factorised vector **kv** stored as a skyline. kdiag holds the locations of the diagonal terms.
sparin_gauss (spabac_gauss)	**kv**, kdiag	Returns the (Gaussian) factorised vector **kv** stored as a skyline. kdiag holds the locations of the diagonal terms.
stability	kv, gv, kdiag, tol, limit, iters, **evec**, **eval**	Returns the smallest eigenvalue **eval** and corresponding eigenvector **evec** of a compressed beam with global stiffness and geometric matrices held in kv and gv respectively. kdiag holds the locations of the diagonal terms, limit is the iteration ceiling and iters is the number of iterations to reach convergence with tolerance tol.

Name	Arguments	Description
`stiff4`	**km**, `coord`, `e`, `v`	Returns the "analytical" stiffness matrix **km** of a general plane strain 4-node quadrilateral element based on 4 Gauss points. `coord` holds the element nodal coordinates, `e` and `v` hold Young's modulus and Poisson's ratio respectively.
`vecmsh`	`loads`, `nf`, `ratmax`, `cutoff`, `g_coord`, `g_num`, `ips`	Generates a PostScript image of the nodal displacement vectors to output channel `ips` with file name `fe95.vec`. `g_coord` and `g_num` hold nodal coordinates and element connectivity. `loads` holds the nodal displacements. `nf` is the nodal freedom array. `ratmax` holds the ratio of the maximum nodal displacement as plotted as a proportion of the longest *x*- or *y*-dimension of the mesh, `cutoff` gives the length of the shortest vector to be plotted as a proportion of the longest.
`vmdpl`	`dee`, `stress` **pl**	Returns the plastic stress–strain matrix **pl** for a von Mises material. `stress` holds the stresses and `dee` holds the elastic stress–strain matrix.
`vmflow`	`stress`, `dsbar`, **vmfl**	Returnst:ıthe von Mises flow vector **vmfl**. `stress` holds the stresses and `dsbar` holds the second deviatoric invariant (2D only).

E

geom Library Subroutines

This Appendix describes the "geometry" subroutines held in a library called geom, which is attached to the main programs described in Chapters 4 to 11 in this book through the command USE geom.

All the subroutines are listed in full on the web site
www.mines.edu/fs_home/vgriffit/4th_ed/source/library/geom

Subroutine descriptions

The following descriptions indicate the geom library subroutines in alphabetic order, together with the meaning of their arguments. Arguments in **bold** are those returned by the subroutine.

Name	Arguments	Description
emb_2d_bc	nx1, nx2, ny1, ny2, **nf**	Returns nodal freedom array **nf** for a 2D embankment analysis with mesh defined by nx1, nx2, ny1 and ny2
emb_2d_geom	iel, nx1, nx2, ny1, ny2, w1, s1, w2, h1, h2, **coord**, **num**	Returns element nodal coordinates **coord** and node numbers **num** for a 2D embankment analysis with geometry and mesh defined by nx1, nx2, ny1, ny2, w1, s1, w2, h1, h2
emb_3d_bc	ifix, nx1, nx2, ny1, ny2, nze **nf**	Returns nodal freedom array **nf** for a 3D embankment analysis with mesh defined by nx1, nx2, ny1, ny2, nze. ifix defines boundary conditions.

© 2004 John Wiley & Sons, Ltd ISBNs: 0-470-84970-3 (PBK); 0-470-84969-X (PPC)

Name	Arguments	Description
`emb_3d_geom`	`iel`, `nx1`, `nx2`, `ny1`, `ny2`, `nze`, `w1`, `s1`, `w2`, `h1`, `h2`, `d1`, **coord**, **num**	Returns element nodal coordinates **coord** and node numbers **num** for a 3D embankment analysis with geometry and mesh defined by `nx1`, `nx2`, `ny1`, `ny2`, `nze`, `w1`, `s1`, `w2`, `h1`, `h2`, `d1`
`fmcoem`	`g_num`, **g_coord**, `fwidth`, `fdepth`, `width`, `depth`, `lnxe`, `lifts`, `fnxe`, `fnye`, `itype`	Returns the global coordinates **g_coord** for a sequential 2D embankment analysis. `g_num` holds the global node numbering. Mesh coordinates `fwidth`, `fdepth`, and `width`, `depth`. Number of columns of element in embankment `lnxe`. Foundation discretisation `fnxe`, `fnye`. Number of lifts `lifts`. Slope element type `itype`.
`fmglem`	`fnxe`, `fnye`, `lnxe`, **g_num**, `lifts`	Returns the global node numbering **g_num** for a sequential embankment analysis. Foundation discretisation `fnxe`, `fnye`. Number of columns of elements in embankment `lnxe`. Number of lifts `lifts`.
`geom_freesurf`	`iel`, `nxe`, `fixed_seep`, `fixed_down`, `down`, `width`, `angs`, `surf`, **coord**, **num**	Returns element nodal coordinates **coord** and node numbers **num** for a 2D free surface analysis. `iel` is the element number. `nxe` is the number of columns of elements. `fixed_seep` is the number of nodes on the seepage surface. `fixed_down` and `down` are the number of nodes and the fixed head on the downstream side. `width` and `angs` are the x-coordinates and inclination angle at the base of the mesh. `surf` holds the y-coordinates of the free surface.

Name	**Arguments**	**Description**
`geom_rect`	`element`, `iel`, `x_coords`, `y_coords`, **coord**, **num**, `dir`	Returns element nodal coordinates **coord** and node numbers **num** for a rectangular mesh of triangles or quadrilaterals. Element type is `element`. `iel` is the element number. `x_coords` and `y_coords` are the *x*- and *y*-coordinates of the mesh. `dir` is the node numbering direction.
`hexahedron_xz`	`iel`, `x_coords`, `y_coords`, `z_coords`, **coord,num**	Returns element nodal coordinates **coord** and node numbers **num** for a cuboidal mesh of hexahedra with nodes numbered in *x*- then *z*- then *y*-.`iel` is the element number. `x_coords`, `y_coords` and `z_coords` are the *x*-, *y*- and *z*-coordinates of the mesh.
`mesh_size`	`element`, `nod`, **nels**, **nn**, `nxe`, `nye`, `nze`	Returns the number of elements **nels** and the number of nodes **nn** in a rectangular mesh of triangles or quadrilaterals. Element type is `element` with `nod` nodes. `nxe`, `nye` and `nze` are the numbers of elements in the *x*-, *y*- and *z*-(3D only) directions.

F

Parallel Library Subroutines

This Appendix describes the additional library subroutines used by the Chapter 12 programs illustrating parallel finite element computations.

All the subroutines are listed in full on the web site `www.mines.edu/fs_home/vgriffit/4th_ed/source/library/parallel` and further information is at `www.parafem.org.uk`.

Subroutine descriptions

The following descriptions indicate the library subroutines in alphabetic order, together with the meaning of their arguments. Arguments in **bold** are those returned by the subroutine.

Name	Arguments	Description
`bcast_inputdata_p???`	`REAL` and `INTEGER` `data to be` broadcast	Broadcasts to all processors in program `p???`
`biot_cube_bc20`	`nxe, nye, nze,` **rest**	Returns restraint array **rest** for numbers of elements in x,y,z of 20/8 node bricks (Biot).
`biot_loading`	`nxe, nze, nle,` **no**, **val**	Returns loaded freedoms **no** and values **val** from number of elements in x and z and number of loaded elements along a side of a square (Biot).

Name	Arguments	Description
`box_bc8`	`nxe`, `nye`, `nze`, **rest**	Returns restraint array **rest** for numbers of elements in x,y,z of 8-node bricks
`calc_nels_pp`	None	Returns number of elements on each parallel processor
`calc_neq_pp`	None	Returns number of equations on each parallel processor
`checon_par`	`xnew_pp`, `x_pp`, `tol`, **converged**, `neq_pp`	Parallel version of `checon`: **converged** returned as `.TRUE.` if difference between `xnew_pp` and `x_pp` less than `tol`: `neq_pp` eqns per processor
`cube_bc20`	`nxe`, `nye`, `nze`, **rest**	Returns restraint array **rest** for numbers of elements in x,y,z of 20-node bricks

`find_g`
`find_g3`
`find_g4`
	`num`, **g**, `rest`	Find steering vector **g** from nodes `num` and restraints `rest`

`find_pe_procs`	None	Returns number of parallel processors being used
`formupvw`	**ke**, `c11`, `c12`, `c21`, `c23`, `c32`, `c24`, `c42`	Returns the element **ke** matrix for coupled 3D Biot consolidation from constituent matrices `c11`, `c12`, `c21`, `c23`, `c32`, `c24`, `c42`
`gather`	`p_pp`, **pmul_pp**	Gathers distributed `p_pp` into **pmul_pp**
`geometry_8bxz`	`ielpe`, `nxe`, `nze`, `aa`, `bb`, `cc`, **coord**, **num**	Finds nodes **num** and nodal coordinates **coord** for a box of 8-node bricks with `nxe` in x and `nze` in z with x,y,z sizes `aa,bb,cc`. `ielpe` is start address per processor
`geometry_20bxz`	`ielpe`, `nxe`, `nze`, `aa`, `bb`, `cc`, **coord**, **num**	Finds nodes **num** and nodal coordinates **coord** for a box of 20-node bricks with `nxe` in x and `nze` in z with x,y,z sizes `aa,bb,cc`. `ielpe` is start address per processor
`g_t_g`	`nod`, `g_t`, **g**	Finds steering vector **g** (Biot) from total vector `g_t`, `nod` nodes

Name	**Arguments**	**Description**
g_t_g_ns	nod, g_t, **g**	Finds steering vector **g** (Navier–Stokes) from total vector g_t, nod nodes
loading	nxe, nze, nle, **no**, **val**	Returns loaded freedoms **no** and values **val** from number of elements in *x* and *z* and number of loaded elements along a side of a square
make_ggl	g_g_pp	Makes the distributed steering vectors
ns_cube_bc20	nxe, nye, nze, **rest**	Returns restraint array **rest** for numbers of elements in *x*,*y*,*z* of 20/8 node bricks
ns_loading	nxe, nye, nze, **no**	Returns "loaded" freedoms **no** for a cuboid with nxe, nye, nze elements in x,y,z
rearrange rearrange_2	**rest**	Reorganise restraint array **rest**
reduce	nn_temp	Returns maximum of a distributed INTEGER variable nn_temp
reindex_fixed_ nodes	ieq_start, no, **no_local_temp**, **num_no**, **no_index_start**	Organises loaded or displaced freedoms in no to locate them in parallel.
scatter	**u_pp**, utemp_pp	Scatters utemp_pp to distributed **u_pp**
vmpl	dee, stress, **pl**	See vmdpl, Appendix D

Author Index

Index compiled by Geraldine Begley

Subject Index

Index compiled by Geraldine Begley