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.

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)

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 `ALLOCATE`d 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
```

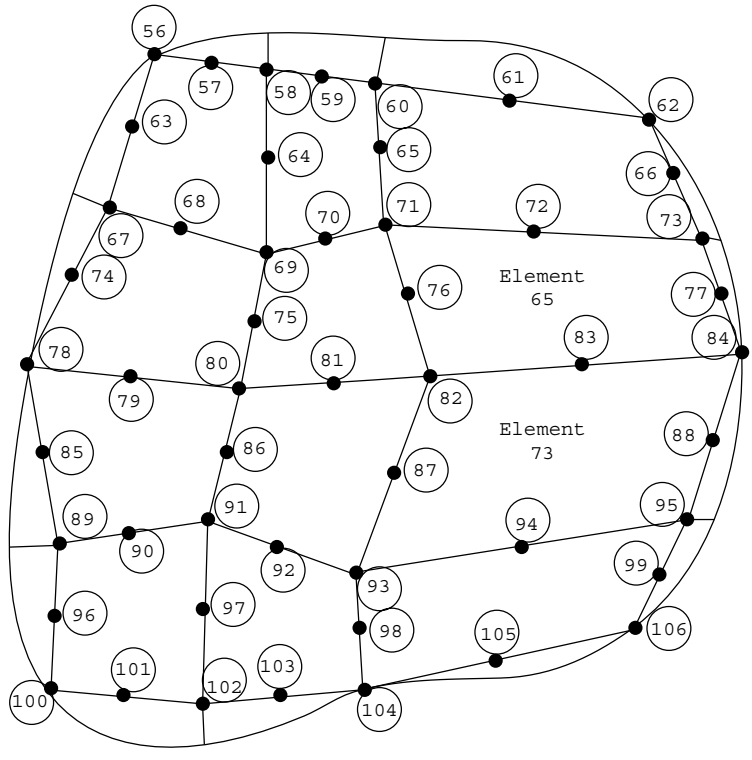


Figure 1.2 Portion of a finite element mesh with node and element numbers

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

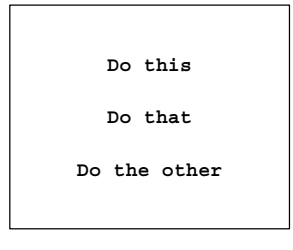
```
PROGRAM test_program_2
 USE library_routines
         .
         .
         .
END PROGRAM test_program_2
```

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.

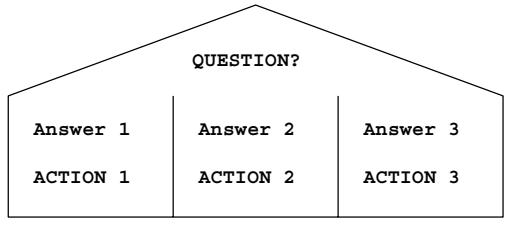
The main features of these charts are:

(i) The block



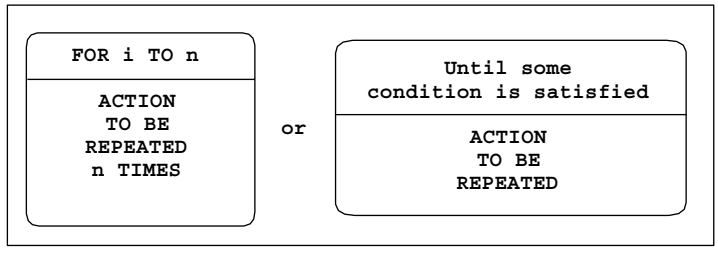
This will be used for the outermost level of each structure chart. Within a block, the indicated actions are to be performed sequentially.

(ii) The choice



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

(iii) The loop



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.

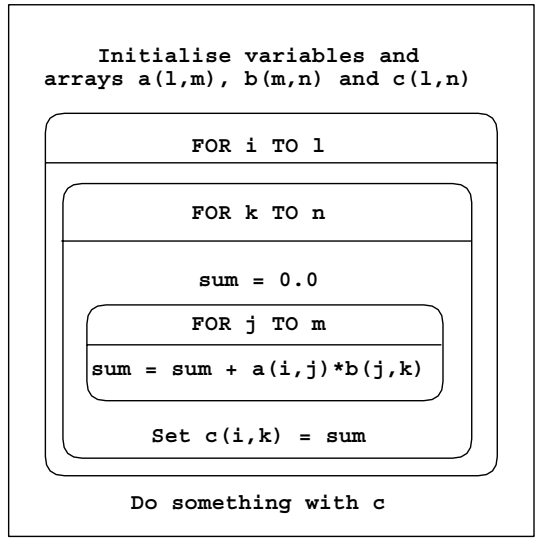


Figure 1.4 Structure chart for matrix multiplication

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.