

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 and scientific 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 and scientific 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 100) and all test programs (numbering over 70) have been verified on a wide range of computers. Efficiency is considered.

The chosen programming language is FORTRAN which remains, overwhelmingly, the most popular language for writing large engineering and scientific programs. Later in this chapter a brief description of the features of FORTRAN which influence the programming of the finite element method will be given. The most recent update of the language was in 2008 (ISO/IEC 1539-1:2010). For parallel environments, MPI has been used, although the programming strategy has also been tested with OpenMP, or a combination of the two.

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. For those who do not have access to the latter and occasionally wish to run large analyses, it is possible to gain access to such facilities on a pay-as-you-go basis through Cloud Computing (see Chapter 12). 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, multi-core, graphics and parallel processors are described later (see Sections 1.4 to 1.7).

1.3 Memory Management

In the programs in this book it will be assumed that sufficient main random access memory 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, 1,000,000 by 10,000. Thus a computer would need to have a main memory of 10^{10} words (tens of Gigabytes) to hold this information, and while some such computers exist, they are comparatively rare. A more typical memory size is of the order of 10^9 words (a Gigabyte).

One strategy to get round this problem is for the programmer to write ‘out-of-memory’ or ‘out-of-core’ 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 virtual 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. 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^{10} 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 10 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 (in a special cache memory, for example).

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 Multi-core Processors

Personal computers from the 1980s onwards originally had one processor with a single central processing unit. Every 18 months or so, manufacturers were able to double the number of transistors on the processor and increase the number of operations that could be performed each second (the clock speed). By the 2000s, miniaturisation of the circuits reached a physical limit in terms of what could be reliably manufactured. Another problem was that it was becoming increasingly difficult to keep these processors cool and energy efficient. These design issues were side-stepped with the development of multi-core processors. Instead of increasing transistor counts and clock speeds, manufacturers began to integrate two or more independent central processing units (cores) onto the same single silicon die or multiple dies in a single chip package. Multi-core processors have gradually replaced single-core processors on all computers over the past 10 years.

The performance gains of multi-core processing depend on the ability of the application to use more than one core at the same time. The programmer needs to write software to execute in parallel, and this is covered later. These modern so-called 'scalar' computers also tend to contain some vector-type hardware. The latest Intel processor has 256-bit vector units on each core, enough to compute four 64-bit floating point operations at

the same time (modest compared with true vector processors). In this book, beginning at Chapter 5, programs which ‘vectorise’ well will be illustrated.

1.6 Co-processors

Co-processors are secondary processors, designed to work alongside the main processor, that perform a specific task, such as manipulating graphics, much faster than the host ‘general-purpose’ processor. The principle of specialisation is similar to vector processing described earlier. Historically, the inclusion of co-processors in computers has come and gone in cycles.

At the time of writing, graphics processing units (GPUs) are a popular way of accelerating numerical computations. GPUs are essentially highly specialised processors with hundreds of cores. They are supplied as plug-in boards that can be added to standard computers. One of the major issues with this type of co-processor is that data needs to be transferred back and forth between the computer’s main memory and the GPU board. The gains in processing speed are therefore greatly reduced if the software implementation cannot minimise or hide memory transfer times. To overcome this, processors are beginning to emerge which bring the graphics processor onto the same silicon die. With multiple cores, a hierarchical memory and special GPU units, these processors are referred to as a ‘system on a chip’ and are the next step in the evolution of modern computers.

There are two main approaches to writing scientific software for graphics processing units: (1) the Open Computing Language (OpenCL) and (2) the Compute Unified Device Architecture (CUDA). OpenCL (<http://www.khronos.org/opencl>) is an open framework for writing software that gives any application access to any vendor’s graphics processing unit, as well as other types of processor. CUDA (<http://developer.nvidia.com/category/zone/cuda-zone>) is a proprietary architecture that gives applications access to NVIDIA hardware only. The use of graphics processing units is covered further in Chapter 12.

1.7 Parallel Processors

In this concept (of which there are many variants) there are several physically distinct processing elements (a few cores in a processor or a lot of multi-core processors in a computer, for example). These processors may also have access to co-processors. Programs and/or data can reside on different processing elements 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 but has not so far been implemented successfully.

For some 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 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 multi-core processors, 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).

The smallest example of a shared memory machine is a multi-core processor which typically has access to a single bank of main memory. In parallel computers comprising many multi-core processors, it is sometimes advantageous to use a hybrid programming strategy whereby OpenMP is used to facilitate communication between local cores (within a single processor) and MPI is used to communicate with remote cores (on other processors).

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 and scientific computations on all machines are usually programmed in ‘high-level’ languages which are intended to be machine-independent. FORTRAN is by far the most widely used language for programming engineering and scientific calculations and in this section a brief overview of FORTRAN will be given with particular reference to features of the language which are useful in finite element computations.

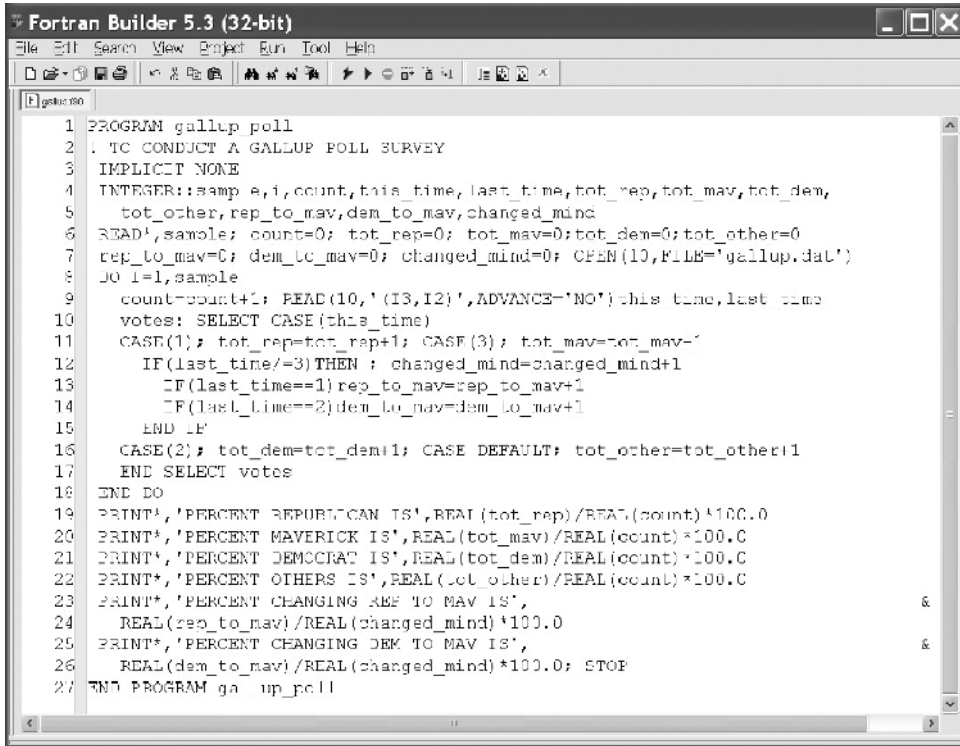
Figure 1.1 shows a typical simple program written in FORTRAN (Smith, 1995). It concerns an opinion poll survey and serves to illustrate the basic structure of the language for those used to other languages.

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.
- 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 a great help in debugging programs.
- Declarations involve the :: double colon convention.
- There are no labelled statements.

1.8.1 Compilers

The human-readable text in Figure 1.1 is turned into computer instructions using a program called a ‘compiler’. There are a number of free compilers available



```

1 PROGRAM gallup_poll
2 . TC CONDUCT A GALLUP POLL SURVEY
3 IMPLICIT NONE
4 INTEGER::samp_e,i,count,this_time,last_time,tot_rep,tot_mav,tot_dem,
5     tot_other,rep_to_mav,dem_to_mav,changed_mind
6 READ',sample; count=0; tot_rep=0; tot_mav=0;tot_dem=0;tot_other=0
7 rep_to_mav=0; dem_to_mav=0; changed_mind=0; OPEN(10,FILE='gallup.dat')
8 DO i=1,sample
9     count=count+1; READ(10,'(I3,I2)',ADVANCE='NO')this_time,last_time
10    votes: SELECT CASE(this_time)
11    CASE(1); tot_rep=tot_rep+1; CASE(3); tot_mav=tot_mav+1
12    IF(last_time/=3)THEN ; changed_mind=changed_mind+1
13    IF(last_time==1)rep_to_mav=rep_to_mav+1
14    IF(last_time==2)dem_to_mav=dem_to_mav+1
15    END IF
16    CASE(2); tot_dem=tot_dem+1; CASE DEFAULT; tot_other=tot_other+1
17    END SELECT votes
18 END DO
19 PRINT*, 'PERCENT REPUBLICAN IS',REAL(tot_rep)/REAL(count)*100.0
20 PRINT*, 'PERCENT MAVERICK IS',REAL(tot_mav)/REAL(count)*100.0
21 PRINT*, 'PERCENT DEMOCRAT IS',REAL(tot_dem)/REAL(count)*100.0
22 PRINT*, 'PERCENT OTHERS IS',REAL(tot_other)/REAL(count)*100.0
23 PRINT*, 'PERCENT CHANGING REP TO MAV IS',
24     REAL(rep_to_mav)/REAL(changed_mind)*100.0
25 PRINT*, 'PERCENT CHANGING DEM TO MAV IS',
26     REAL(dem_to_mav)/REAL(changed_mind)*100.0; STOP
27 END PROGRAM gallup_poll

```

Figure 1.1 A typical program written in FORTRAN

that are suitable for students, such as G95 (www.g95.org) and GFORTRAN (<http://gcc.gnu.org/fortran/>). Commercial FORTRAN compilers used in the book include those supplied by Intel, Cray, NAG and the Portland Group. When building an application on a supercomputer, use of the compiler provided by the vendor is highly recommended. These typically generate programs that make better use of the target hardware than free versions.

Figure 1.1 shows a Windows-based programming environment in which FORTRAN programs can be written, compiled and executed with the help of an intuitive graphical user interface. FORTRAN programs can also be written using a text editor and compiled using simple commands in a Windows or Linux terminal. An example of how to compile at the ‘command line’ is shown below. The compiler used is G95.

```

g95 -c hello.f90          Creates an object file named hello.o
g95 -o hello hello.f90   Compiles and links to create the executable hello

```

1.8.2 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 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 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, d4=4.0_iwp, penalty=1.0E20_iwp
```

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

1.8.3 Conditions

There are two basic structures for conditional statements in FORTRAN 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 below can be used:

```

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

```

1.8.4 Loops

There are two constructs in `FORTRAN` 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

Since the 1990 revision, FORTRAN has allowed ‘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
  INTEGER, PARAMETER::iwp=SELECTED_REAL_KIND(15)
  ! declare variable space for two dimensional array a
  REAL(iwp), ALLOCATABLE::a(:, :)
  REAL(iwp)::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 ! reads array a column by column
  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 language. 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 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 the finite element method and other numerical methods such as the boundary element method (Beer *et al.*, 2008). 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

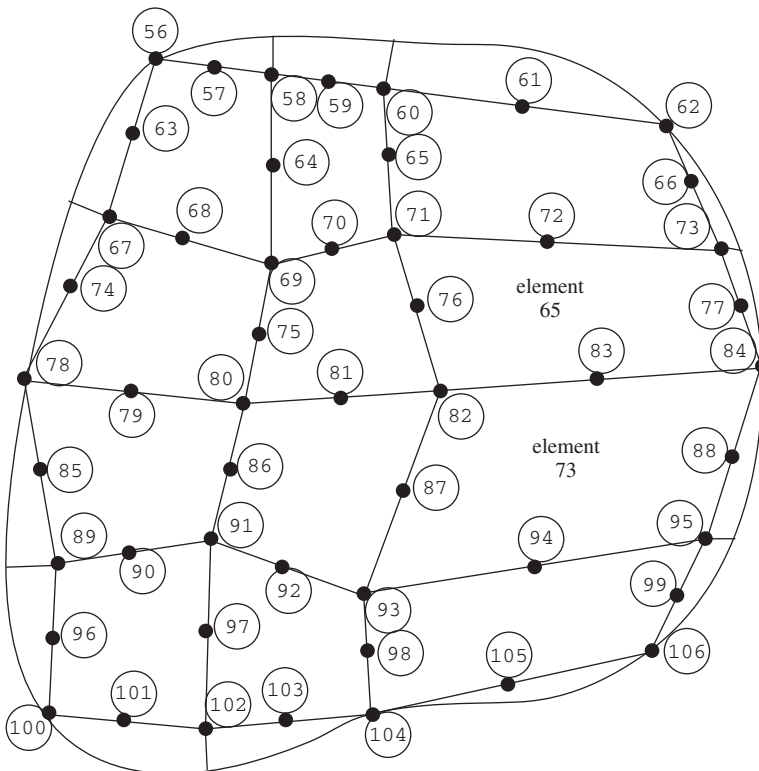


Figure 1.2 Portion of a finite element mesh with node and element numbers

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
```

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 eleven through thirteen 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

Whilst simple operations on whole arrays such as addition, multiplication by a scalar and so on are easily carried out in FORTRAN it must be noted that 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 subsection.

1.9.7 Intrinsic Procedures for Arrays

To supplement whole-array arithmetic operations, FORTRAN 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. 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.

```

In the event of array *a* having more than one dimension, *MAXLOC* and *MINLOC* return the appropriate number of integers (row, column, etc.) pointing to the required location.

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 multi-dimensional 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) - 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 Modules

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 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.

Support for mixed-language programming has been added to the latest versions of FORTRAN (2003 onwards), enabling interoperability between FORTRAN and C. This has been made easier by the introduction of an intrinsic module that helps programmers ensure that a variable of particular type and kind used in FORTRAN maps to a variable of the

same type and kind in C. This is invoked by adding the following to the FORTRAN program and declaring the variables as specified by the ISO standard:

```
USE, INTRINSIC :: ISO_C_BINDING
```

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 one or two common subroutine libraries which contain subroutines and functions. In the simplest implementation of FORTRAN 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 sublibrary 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 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.

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 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 as shown in Figure 1.3, 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 as shown in Figure 1.4.

(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) as shown in Figure 1.5.

Using this notation, a matrix multiplication program would be represented as shown in Figure 1.6. The nested nature of a typical program can be seen quite clearly.

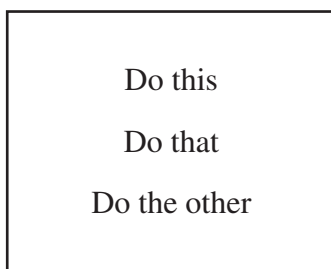


Figure 1.3 The block

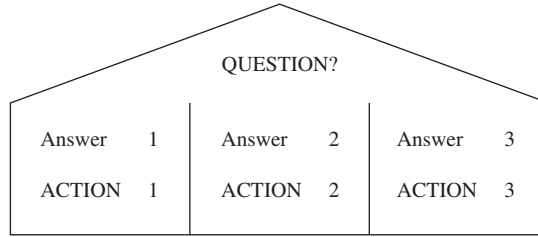


Figure 1.4 The choice

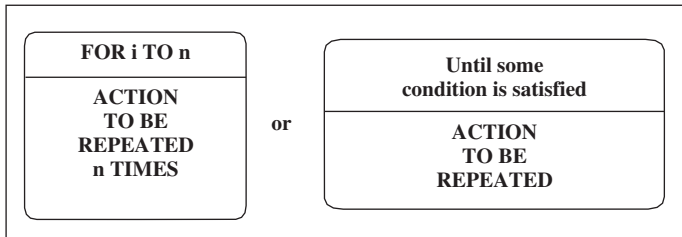


Figure 1.5 The loop

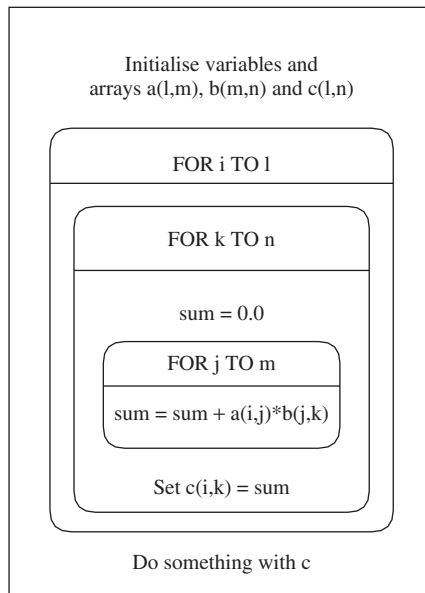


Figure 1.6 Structure chart for matrix multiplication

1.10 Third-party Libraries

The programs and libraries provided in this book are mostly self-contained and have been written by the authors. This philosophy clarifies the book as a teaching text and simplifies the learning process. However, there are many third-party libraries that can be used either to extend the capabilities of the software provided or to improve its execution speed. Some examples are provided in the following sections. External subprograms used in the book are listed in Appendix G.

1.10.1 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` to multiply two matrices together as described in Figure 1.6. 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.

An alternative is to use ‘BLAS’ or Basic Linear Algebra Subroutine Libraries (e.g., Dongarra and Walker, 1995). Most vendors provide a BLAS maths library tuned for their hardware. Special versions for graphics processing units, such as CUBLAS (www.developer.nvidia.com/cublas) and MAGMA (Agullo *et al.*, 2009), are also available.

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)
```

as will be shown in Program 12.1. However, very significant gains in processing speed can be achieved as reported in Chapters 5 and 12.

1.10.2 Maths Libraries

There are a large number of commercial and freely available libraries which contain mathematical and statistical algorithms. NAG Ltd. provides a library with over 1,700 fully documented and tested routines (www.nag.co.uk). The UK Rutherford Appleton Laboratory develops the Harwell Scientific Library (www.hsl.rl.ac.uk), a collection of packages for large-scale scientific computation that have been continually updated and improved since 1963. For eigenvalue problems, an excellent resource is the ARPACK library that is used in Chapter 10 (www.caam.rice.edu/software/ARPACK).

1.10.3 User Subroutines

Most commercial finite element packages provide interfaces for users to incorporate subroutines they have written themselves. ‘User subroutine’ interfaces are provided for features that are not yet implemented in the package, such as special element types, new models of material behaviour and faster equation solvers. It is very straightforward to use these subroutines to extend the capabilities of the programs described in this book. A simple example of using an ABAQUS (www.3ds.com) `umat` (User MATERIAL) is shown in Chapter 5. In this case, the `umat` is written in an old version of FORTRAN, a specific requirement for its use in ABAQUS.

1.10.4 MPI Libraries

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

```
CALL MPI_BCAST(rest,buf,MPI_INTEGER,npes-1,MPI_COMM_WORLD,ier)
```

‘broadcasts’ the array `rest` of size `buf` (number of restraints `nr` multiplied by degrees of freedom `nodof+1`) 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 MPI library is left to Chapter 12. A recommended implementation of MPI, needed to run MPI-based programs, is OpenMPI (www.open-mpi.org).

1.11 Visualisation

It is good practice to inspect finite element models before analysis using a visualisation tool in order to check the quality of the mesh and ensure that the loading and boundary conditions have been correctly applied. The same tool can be used after the analysis to plot, for example, the deformed mesh and contours of derived quantities such as stress and strain. Two visualisation strategies are adopted here. The first uses subroutines to conveniently generate PostScript images as direct output from the programs:

```
SUBROUTINE mesh          ! Image of undeformed mesh
SUBROUTINE dismsh       ! Image of deformed mesh
SUBROUTINE vecmsh       ! Image of nodal displacement vectors
SUBROUTINE contour      ! Image of contours of nodal values
```

The second uses a third-party visualisation tool, ParaView, that can be freely downloaded (www.paraview.org). To use ParaView, the finite element programs need to output data in a format that ParaView supports. Here the following subroutines are provided that output geometry and results in the Enight Gold format:

```
SUBROUTINE mesh_ensi     ! Undeformed mesh files
SUBROUTINE dismsh_ensi  ! Displacement file(s)
```

For Program 5.6, say, a typical set of output files generated by these two subroutines will contain

```
p56.ensi.case           ! a descriptive control file
p56.ensi.geo           ! geometry and element steering array
p56.ensi.ndlds         ! nodal loads
p56.ensi.ndbnd         ! restrained loads
p56.ensi.dis*****    ! nodal displacements,
                       ! where ***** is the step number
```

Basic instructions for using ParaView, in Windows, to create typical plots are provided in the following sections. ParaView is a very powerful visualisation tool and for more advanced usage, it is recommended that the reader consults the ParaView documentation. For very large data sets (e.g., Chapter 12), ParaView can be run in parallel mode. These instructions are for ParaView version 3.98.0.

1.11.1 Starting ParaView

After downloading from the website and following the installation instructions, the ParaView application should appear in the Start menu. When launched, the ParaView display is initially split into four main areas as shown in Figure 1.7. Across the top are the menus and toolbars. On the left is the Pipeline Browser, which shows the loaded model and any objects derived from the model. Below that are the Properties

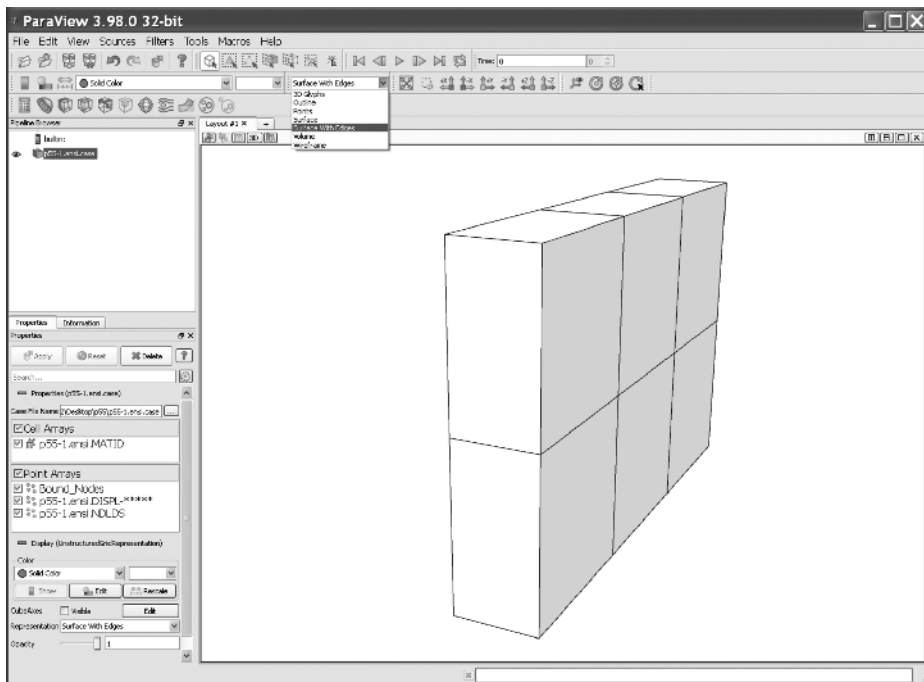


Figure 1.7 Undeformed mesh loaded into ParaView

and Information tabs, which show details about the model and a list of parameters the user can modify. Finally, the main area is the Viewer Window that shows the current view of the model.

To load a finite element model, use `File > Open`, navigate to the directory where the data set is located and select the case file (e.g., `p56.ensi.case`). Some information will appear in the Pipeline Browser and Properties tab. In the Properties tab, there will be a number of named variables that belong to the data set. At this stage, all of these should be checked and the Apply button should be applied. The model should now appear in the main viewer window. To view the undeformed mesh, click on the Representation dropdown box at the bottom of the Properties tab and select Surface with Edges. There is also a dropdown box that performs the same action in the toolbars that appear at the top of the ParaView display.

The model can be rotated by clicking, holding and dragging the mouse. Zooming in and out is done by holding down the control (`ctrl`) button on the keyboard and the left mouse button at the same time.

1.11.2 Display Restrained Nodes

The scalar value in the EnSight file `p56.ensi.NDBND` is derived from the following bitwise function: $\text{value} = !z*4 + !y*2 + !x$, where x , y and z are either 1 – the node is constrained in that axis, or 0 – the node is free to move in that axis (note this is the opposite of the convention used in the book). A bound node of 0 0 0 (book convention) or 1 1 1 (EnSight convention) will result in a value of 7 ($1*4 + 1*2 + 1*1$) in the EnSight file, denoting that the node is restrained in all three axes. Further examples of this coding convention for restraints are given below:

Book	(EnSight)	! Integer restraint code in p56.ensi.NDBND
0 0 0	(1 1 1)	! $1*4 + 1*2 + 1*1 = 7$
1 0 0	(0 1 1)	! $0*4 + 1*2 + 1*1 = 3$
0 1 0	(1 0 1)	! $1*4 + 0*2 + 1*1 = 5$

To view the restrained nodes as shown in Figure 1.8, first select the `p56.ensi.case` object in the Pipeline Browser. Now add a Calculator filter by selecting the calculator icon. In the Properties tab, change the Result Array Name to `isConstrained`. Just below that is a type-in area where a calculator function can be entered. Type `if(restraint>6,1,0)` and press Apply. This simply generates a 0 or 1 value to denote if the constraint 0 0 0 (book convention) is present. A similar strategy is taken to view other types of restraint. The `if` syntax is `(condition, true-value, false-value)`. With the Calculator object selected in the Pipeline Browser, apply a glyph filter using the Glyph icon. In the Properties tab, change the Scale Mode to scalar, select the Scalars dropdown to be `isConstrained`, select Glyph Type as Sphere and press Apply.

Some sphere glyphs should now appear but they may be too large. The size can be changed by entering a new value in the Radius box in the Sphere panel on the Properties tab. To accept the changes, press Apply. In the Properties tab of the Glyph object, set Color to `restraint`, then press the Edit colour map button immediately below. Choose a colour scheme. In Figure 1.8, a greyscale colour scheme has been selected.

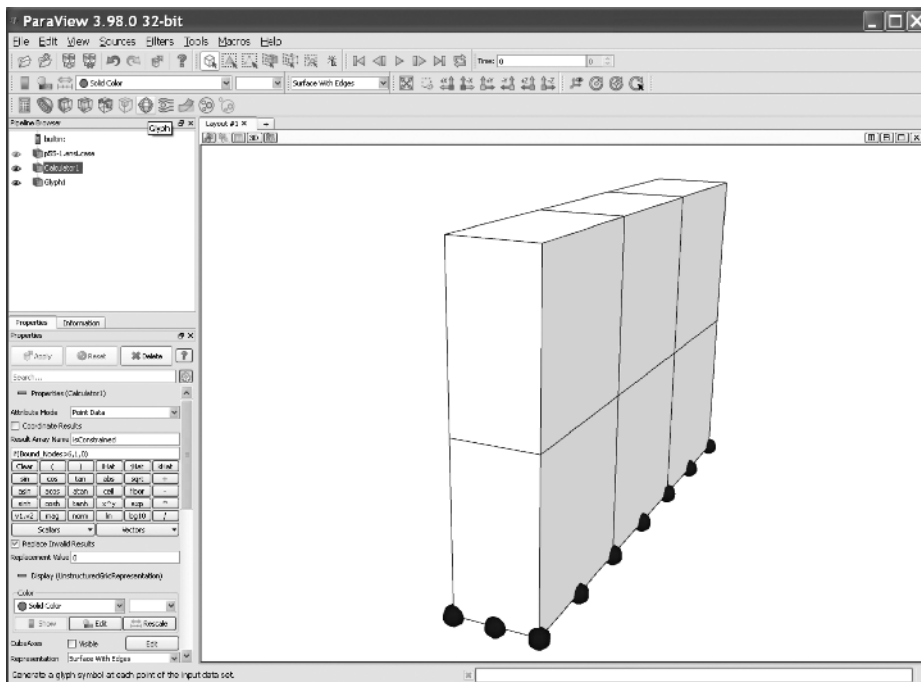


Figure 1.8 Restrained nodes

1.11.3 Display Applied Loads

To view the applied loads, as shown in Figure 1.9, click on the `p56.ensi.case` file in the Pipeline Browser and then select a glyph filter using the Glyph icon. In the Properties tab, change the Scale Mode to Vector, set Vectors to load, select Glyph Type as Arrow and press Apply. Some arrows should appear in the Viewer Window. The arrows are scaled by the magnitude of the vector, so zero and low-magnitude glyphs may not be visible. It can be difficult to see arrows clearly when the model is obscuring them. In the Pipeline Browser, click on the eye icon to the left of the Calculator1 object to ensure it is greyed out. Now select the `p56.ensi.case` file and ensure the eye icon is darkened (the icon toggles between light and dark to indicate whether that object is displayed in the Viewer Window). In the Properties tab, modify the Representation settings so that the model is represented as a Wireframe. Another option is to select Surface and set the Opacity to 0.10 to make the model semi-transparent.

1.11.4 Display Deformed Mesh

The results of `p56` include a set of nodal displacements. This deformation can be applied and visualised directly as shown in Figure 1.10. With the `p56.ensi.case` file selected in the Pipeline Browser, click on the Warp By Vector icon (bendy green bar). This is situated above the Pipeline Browser. A new object will now appear in the Pipeline Browser, called `WarpByVector1`. Select the new `WarpByVector1`

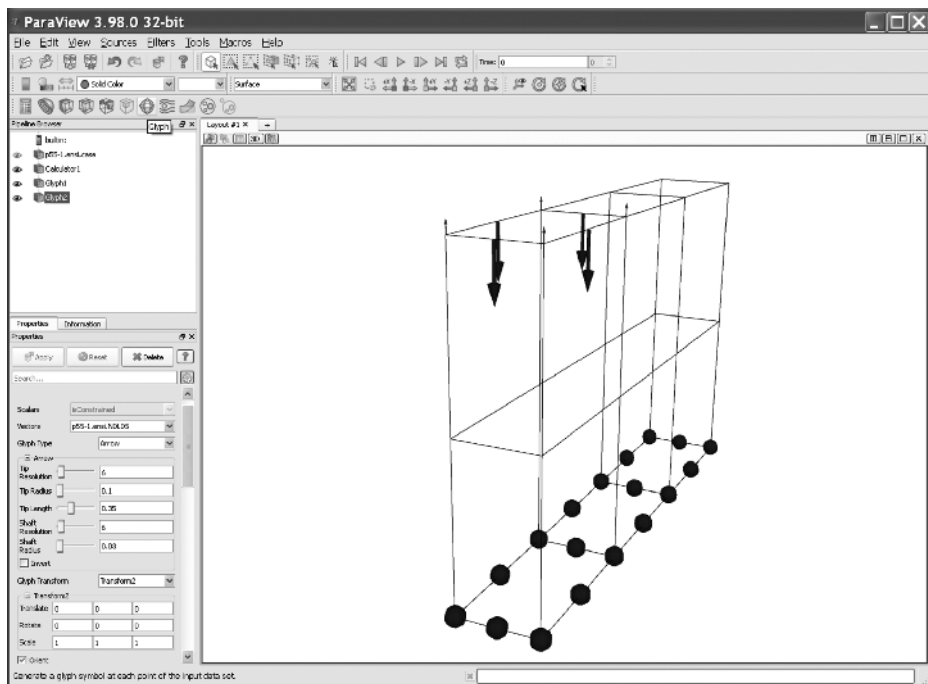


Figure 1.9 Nodal force vectors

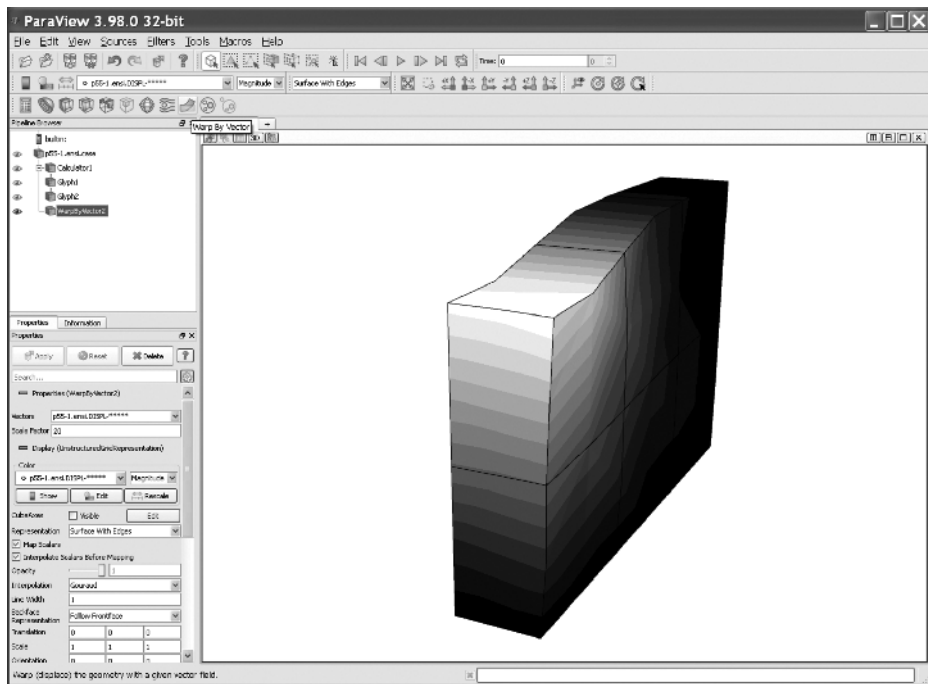


Figure 1.10 Deformed mesh

object in the Pipeline Browser. In its Properties tab, check that the displacement variable is selected in the Vectors dropdown menu and then press the green Apply button. If the displacement is very small and the deformation is hardly noticeable, the Scale Factor can be modified to exaggerate the displacements.

1.12 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. 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 <http://parafem.org.uk>.

Using this philosophy, user libraries containing over 100 subroutines and functions have been assembled, together with over 70 example programs which access them. These programs and subroutines are written in a reasonably 'structured' style, and can be downloaded from the internet at www.mines.edu/~vgriffit/5th_ed. Versions are at present available for all the common machine ranges and FORTRAN compilers listed in Section 1.8.1. The downloadable software includes the parallel library, which consists of more than 20 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 libraries 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.

Chapters 4 to 11 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, bioengineering, 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 field problems (e.g., fluid or heat flow), while transient states with inclusion of transport phenomena (diffusion with convection) 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. A final program in Chapter 12 illustrates the use of GPUs.

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

- Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, *et al.* 2009 Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Physics: Conference Series* **180**.
- Beer G, Smith IM and Duenser C 2008 *The Boundary Element Method with Programming*. Springer, London.
- Dijkstra EW 1976 *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ.
- Dongarra JJ and Walker DW 1995 Software libraries for linear algebra computations on high performance computers. *Siam Rev* **37**(2), 151–180.
- 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. 400–409.
- Smith IM 1995 *Programming in Fortran 90*. John Wiley & Sons, Chichester.
- 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.