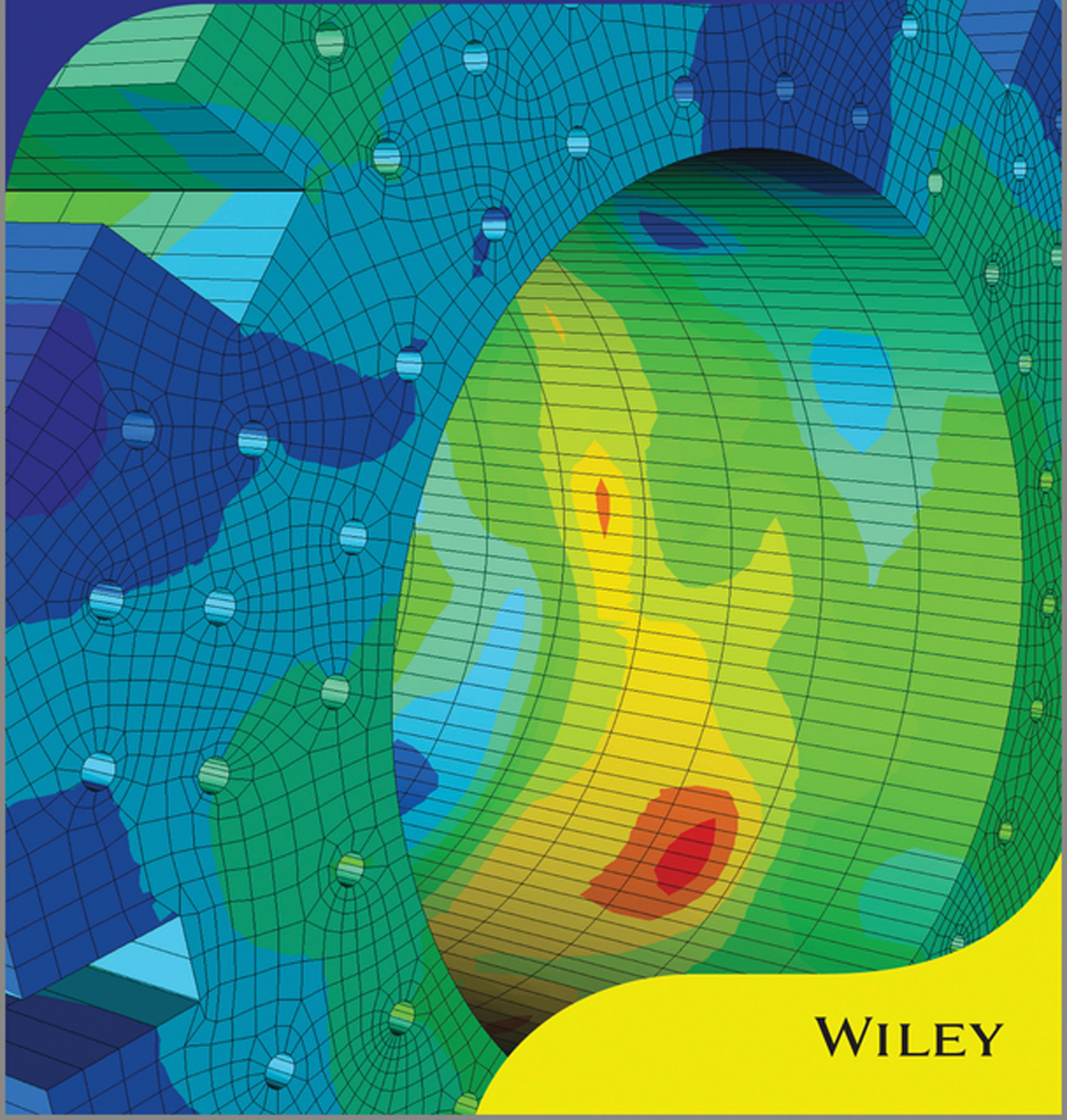
Fifth Edition

Programming the Finite Element Method

I. M. Smith, D. V. Griffiths and L. Margetts



WILEY

PROGRAMMING THE FINITE ELEMENT METHOD

PROGRAMMING THE FINITE ELEMENT METHOD

Fifth Edition

I. M. Smith
University of Manchester, UK

D. V. Griffiths
Colorado School of Mines, USA

L. Margetts
University of Manchester, UK

WILEY

Library of Congress Cataloguing-in-Publication Data

Smith, I. M. (Ian Moffat), 1940- author.
Programming the finite element method. – Fifth edition/Ian M. Smith, D. Vaughan Griffiths, Lee Margetts.
pages cm
Includes bibliographical references and index.
ISBN 978-1-119-97334-8 (hardback)
1. Finite element method–Data processing. 2. Engineering–Data processing. 3. FORTRAN 2003
(Computer program language) I. Griffiths, D. V., author. II. Margetts, Lee, author. III. Title.
TA347.F5S64 2014
620.001'51825–dc23

2013019445

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

ISBN: 978-1-119-97334-8

Set in 10/12pt Times by Laserwords Private Limited, Chennai, India.

1 2014

Contents

Preface to Fifth Edition

This edition maintains the successful theme of previous editions, namely a modular programming style which leads to concise, easy to read computer programs for the solution of a wide range of problems in engineering and science governed by partial differential equations.

The programming style has remained essentially the same despite huge advances in computer hardware. Readers will include beginners, making acquaintance with the finite element method for the first time, and specialists solving very large problems using the latest generation of parallel supercomputers.

In this edition special attention is paid to interfacing with other open access software, for example ParaView for results visualisation, ABAQUS user subroutines for a range of material constitutive models, ARPACK for large eigenvalue analyses, and METIS for mesh partitioning.

Chapter 1 has been extensively rewritten to take account of rapid developments in computer hardware, for example the availability of GPUs and cloud computing environments. In Chapters 2 to 11 numerous additions have been made to enhance analytical options, for example new return algorithms for elastoplastic analyses, more general boundary condition specification and a complex response option for dynamic analyses.

Chapter 12 has been updated to illustrate the rapidly advancing possibilities for finite element analyses in parallel computing environments. In the fourth edition the maximum number of parallel ‘processes’ used was 64 whereas in this edition the number has increased to 64,000. The use of GPUs to accelerate computations is illustrated.

Acknowledgements

The authors wish to acknowledge the contributions of a number of individuals and organizations. The support of the Australian Research Council Centre of Excellence for Geotechnical Science and Engineering (CGSE) at the University of Newcastle NSW is recognised, and particularly Jinsong Huang, who contributed to the development and validation of several of the new and modified programs in Chapters 6, 8 and 9. Louise Lever (University of Manchester), one of the principal ParaFEM developers, provided expertise in the use of ParaView for Chapters 1, 5, 6 and 12 and set up the community building website http://parafem.org.uk.

There were many contributions to Chapter 12. Llion Evans, Paul Mummery, Philip Manning, Graham Hall and Dimitris Christias (all University of Manchester) provided scientific case studies. Florent Lebeau and Francois Bodin (CAPS Entreprise) evaluated the use of GPUs and Philippe Young (Simpleware Ltd) provided support in image-based modelling.

Benchmarking of the programs in Chapter 12 was carried out using supercomputers hosted by the UK National High Performance Computing Service "HECToR" (e107, e254) and the UK Regional Service "N8 HPC" (EP/K000225/1). The EU FP7 project "Venus-C" and Barcelona Supercomputing Center (Spain) provided access, resources and training to use Microsoft Azure.

The authors would also like to thank family members for their support during preparation of the book, including Valerie Griffiths, Laura Sanchez and Nathan Margetts.

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.

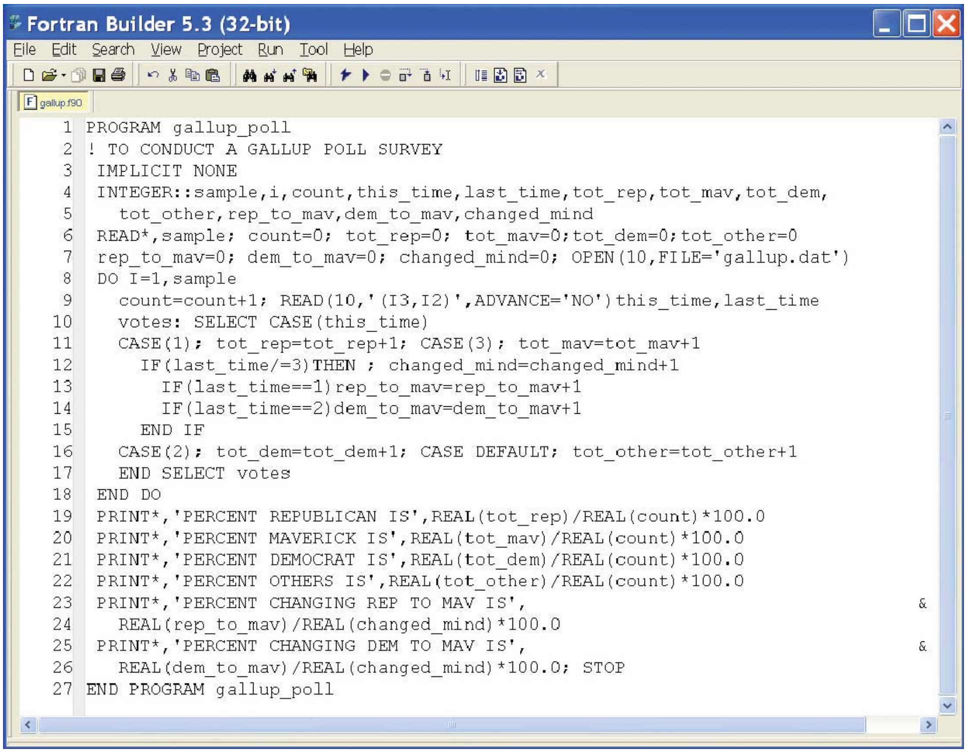
Figure 1.1 shows a typical simple program written in FORTRAN (Smith, 1995). It concerns an opinion poll survey and serves to illustrate the basic structure of the language for those used to other languages.

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


Fortran Builder 5.3 (32-bit)
File Edit Search View Project Run Tool Help
gallup.f90

```fortran
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_dem)/REAL(count)*100.0
 PRINT*,'PERCENT OTHERS IS',REAL(tot_other)/REAL(count)*100.0
 PRINT*,'PERCENT CHANGING REP TO MAV IS',                              &
   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

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,ALLOCATABLE(iwp)::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

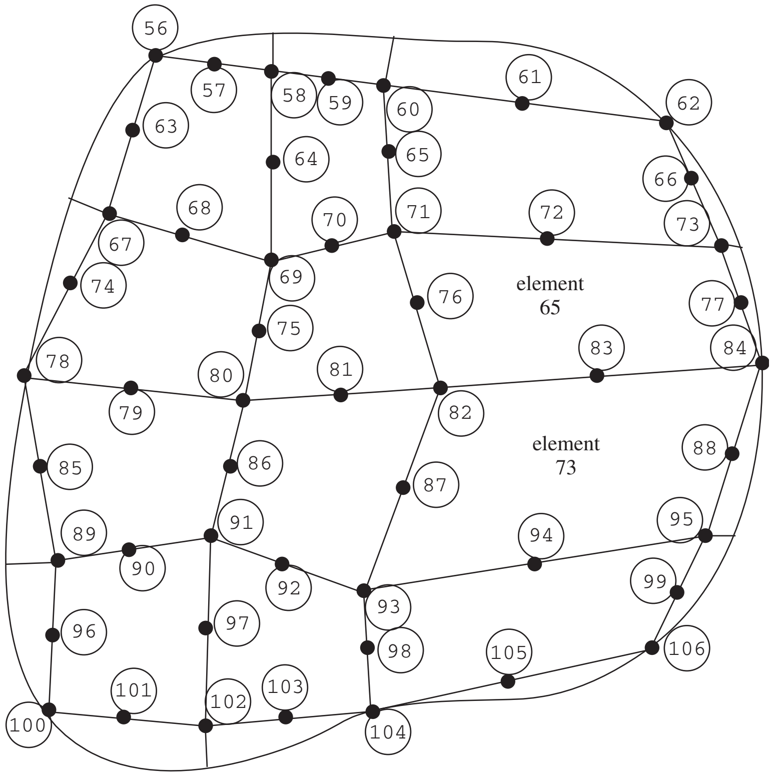


Figure 1.2 Portion of a finite element mesh with node and element numbers