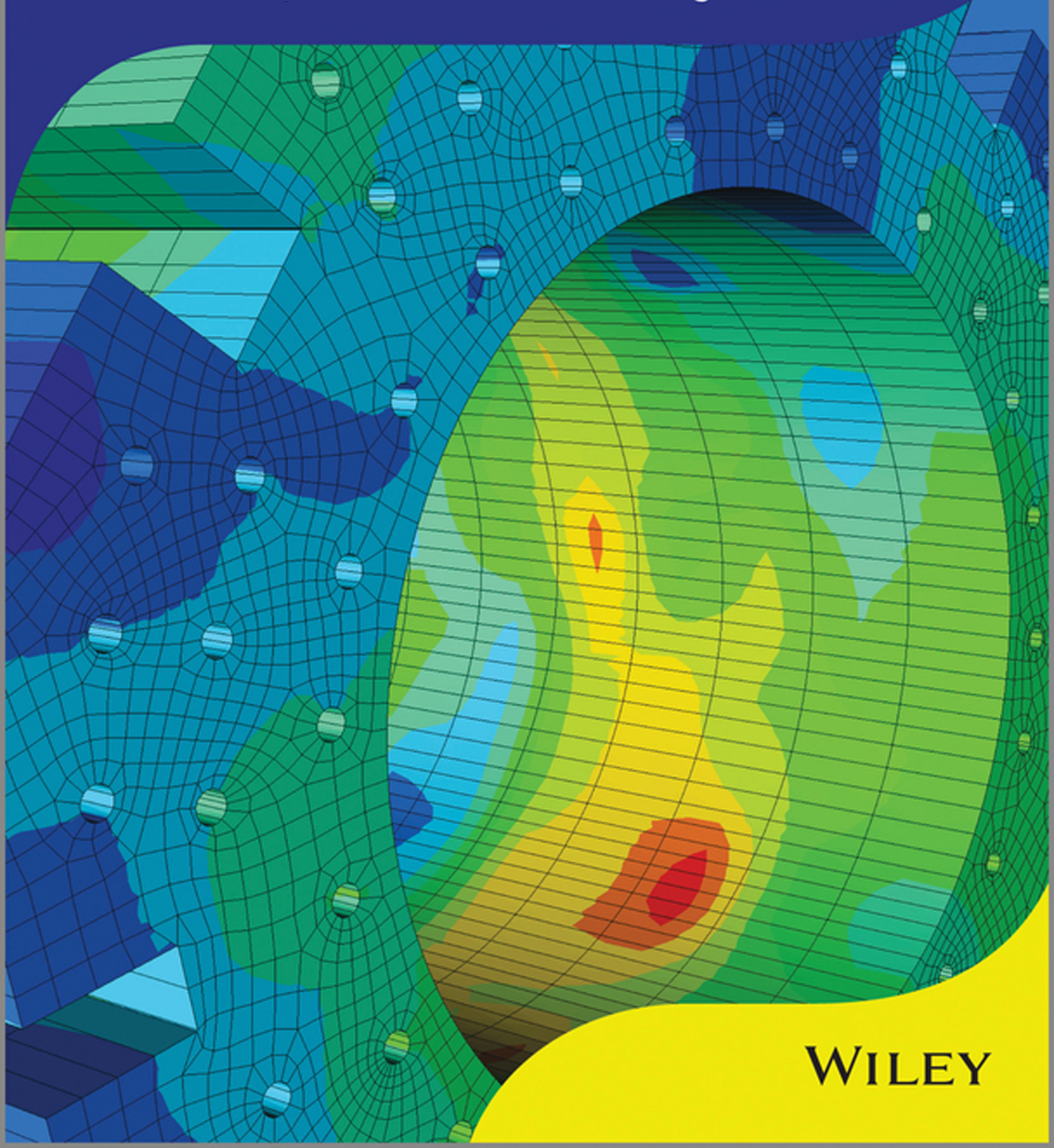


Fifth Edition

# 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**

This edition first published 2014  
© 2014 John Wiley & Sons Ltd

*Registered office*

John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

For details of our global editorial offices, for customer services and for information about how to apply for permission to reuse the copyright material in this book please see our website at [www.wiley.com](http://www.wiley.com).

The right of the author to be identified as the author of this work has been asserted in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by the UK Copyright, Designs and Patents Act 1988, without the prior permission of the publisher.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The publisher is not associated with any product or vendor mentioned in this book.

**Limit of Liability/Disclaimer of Warranty:** While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. It is sold on the understanding that the publisher is not engaged in rendering professional services and neither the publisher nor the author shall be liable for damages arising herefrom. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

*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

<b>Preface to Fifth Edition</b>	<b>xv</b>
<b>Acknowledgements</b>	<b>xvii</b>
<b>1 Preliminaries: Computer Strategies</b>	<b>1</b>
1.1 Introduction	1
1.2 Hardware	2
1.3 Memory Management	2
1.4 Vector Processors	3
1.5 Multi-core Processors	3
1.6 Co-processors	4
1.7 Parallel Processors	4
1.8 Applications Software	5
1.8.1 <i>Compilers</i>	5
1.8.2 <i>Arithmetic</i>	6
1.8.3 <i>Conditions</i>	7
1.8.4 <i>Loops</i>	8
1.9 Array Features	9
1.9.1 <i>Dynamic Arrays</i>	9
1.9.2 <i>Broadcasting</i>	9
1.9.3 <i>Constructors</i>	9
1.9.4 <i>Vector Subscripts</i>	10
1.9.5 <i>Array Sections</i>	11
1.9.6 <i>Whole-array Manipulations</i>	11
1.9.7 <i>Intrinsic Procedures for Arrays</i>	11
1.9.8 <i>Modules</i>	12
1.9.9 <i>Subprogram Libraries</i>	13
1.9.10 <i>Structured Programming</i>	15
1.10 Third-party Libraries	17
1.10.1 <i>BLAS Libraries</i>	17
1.10.2 <i>Maths Libraries</i>	17
1.10.3 <i>User Subroutines</i>	18
1.10.4 <i>MPI Libraries</i>	18
1.11 Visualisation	18
1.11.1 <i>Starting ParaView</i>	19
1.11.2 <i>Display Restrained Nodes</i>	20

1.11.3	<i>Display Applied Loads</i>	21
1.11.4	<i>Display Deformed Mesh</i>	21
1.12	Conclusions	23
	References	24
<b>2</b>	<b>Spatial Discretisation by Finite Elements</b>	<b>25</b>
2.1	Introduction	25
2.2	Rod Element	25
2.2.1	<i>Rod Stiffness Matrix</i>	25
2.2.2	<i>Rod Mass Element</i>	28
2.3	The Eigenvalue Equation	28
2.4	Beam Element	29
2.4.1	<i>Beam Element Stiffness Matrix</i>	29
2.4.2	<i>Beam Element Mass Matrix</i>	31
2.5	Beam with an Axial Force	31
2.6	Beam on an Elastic Foundation	32
2.7	General Remarks on the Discretisation Process	33
2.8	Alternative Derivation of Element Stiffness	33
2.9	Two-dimensional Elements: Plane Stress	35
2.10	Energy Approach and Plane Strain	38
2.10.1	<i>Thermoelasticity</i>	39
2.11	Plane Element Mass Matrix	40
2.12	Axisymmetric Stress and Strain	40
2.13	Three-dimensional Stress and Strain	42
2.14	Plate Bending Element	44
2.15	Summary of Element Equations for Solids	47
2.16	Flow of Fluids: Navier–Stokes Equations	47
2.17	Simplified Flow Equations	50
2.17.1	<i>Steady State</i>	51
2.17.2	<i>Transient State</i>	53
2.17.3	<i>Convection</i>	53
2.18	Further Coupled Equations: Biot Consolidation	54
2.19	Conclusions	56
	References	56
<b>3</b>	<b>Programming Finite Element Computations</b>	<b>59</b>
3.1	Introduction	59
3.2	Local Coordinates for Quadrilateral Elements	59
3.2.1	<i>Numerical Integration for Quadrilaterals</i>	61
3.2.2	<i>Analytical Integration for Quadrilaterals</i>	63
3.3	Local Coordinates for Triangular Elements	64
3.3.1	<i>Numerical Integration for Triangles</i>	65
3.3.2	<i>Analytical Integration for Triangles</i>	65
3.4	Multi-Element Assemblies	66
3.5	‘Element-by-Element’ Techniques	68
3.5.1	<i>Conjugate Gradient Method for Linear Equation Systems</i>	68
3.5.2	<i>Preconditioning</i>	69

3.5.3	<i>Unsymmetric Systems</i>	70
3.5.4	<i>Symmetric Non-Positive Definite Equations</i>	71
3.5.5	<i>Eigenvalue Systems</i>	71
3.6	<i>Incorporation of Boundary Conditions</i>	72
3.6.1	<i>Convection Boundary Conditions</i>	74
3.7	<i>Programming using Building Blocks</i>	75
3.7.1	<i>Black Box Routines</i>	76
3.7.2	<i>Special Purpose Routines</i>	77
3.7.3	<i>Plane Elastic Analysis using Quadrilateral Elements</i>	77
3.7.4	<i>Plane Elastic Analysis using Triangular Elements</i>	81
3.7.5	<i>Axisymmetric Strain of Elastic Solids</i>	82
3.7.6	<i>Plane Steady Laminar Fluid Flow</i>	83
3.7.7	<i>Mass Matrix Formation</i>	83
3.7.8	<i>Higher-Order 2D Elements</i>	84
3.7.9	<i>Three-Dimensional Elements</i>	86
3.7.10	<i>Assembly of Elements</i>	90
3.8	<i>Solution of Equilibrium Equations</i>	95
3.9	<i>Evaluation of Eigenvalues and Eigenvectors</i>	96
3.9.1	<i>Jacobi Algorithm</i>	96
3.9.2	<i>Lanczos and Arnoldi Algorithms</i>	98
3.10	<i>Solution of First-Order Time-Dependent Problems</i>	99
3.11	<i>Solution of Coupled Navier–Stokes Problems</i>	103
3.12	<i>Solution of Coupled Transient Problems</i>	104
3.12.1	<i>Absolute Load Version</i>	105
3.12.2	<i>Incremental Load Version</i>	106
3.13	<i>Solution of Second-Order Time-Dependent Problems</i>	106
3.13.1	<i>Modal Superposition</i>	107
3.13.2	<i>Newmark or Crank–Nicolson Method</i>	109
3.13.3	<i>Wilson’s Method</i>	110
3.13.4	<i>Complex Response</i>	111
3.13.5	<i>Explicit Methods and Other Storage-Saving Strategies</i>	112
	<i>References</i>	113
<b>4</b>	<b>Static Equilibrium of Structures</b>	<b>115</b>
4.1	<i>Introduction</i>	115
	Program 4.1 One-dimensional analysis of axially loaded elastic rods using 2-node rod elements	116
	Program 4.2 Analysis of elastic pin-jointed frames using 2-node rod elements in two or three dimensions	121
	Program 4.3 Analysis of elastic beams using 2-node beam elements (elastic foundation optional)	127
	Program 4.4 Analysis of elastic rigid-jointed frames using 2-node beam/rod elements in two or three dimensions	133
	Program 4.5 Analysis of elastic–plastic beams or frames using 2-node beam or beam/rod elements in one, two or three dimensions	141
	Program 4.6 Stability (buckling) analysis of elastic beams using 2-node beam elements (elastic foundation optional)	150

	Program 4.7 Analysis of plates using 4-node rectangular plate elements. Homogeneous material with identical elements. Mesh numbered in $x$ - or $y$ -direction	153
4.2	Conclusions	157
4.3	Glossary of Variable Names	157
4.4	Exercises	159
	References	168
<b>5</b>	<b>Static Equilibrium of Linear Elastic Solids</b>	<b>169</b>
5.1	Introduction	169
	Program 5.1 Plane or axisymmetric strain analysis of a rectangular elastic solid using 3-, 6-, 10- or 15-node right-angled triangles or 4-, 8- or 9-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction	170
	Program 5.2 Non-axisymmetric analysis of a rectangular axisymmetric elastic solid using 8-node rectangular quadrilaterals. Mesh numbered in $r$ - or $z$ -direction	184
	Program 5.3 Three-dimensional analysis of a cuboidal elastic solid using 8-, 14- or 20-node brick hexahedra. Mesh numbered in $xz$ -planes then in the $y$ -direction	191
	Program 5.4 General 2D (plane strain) or 3D analysis of elastic solids. Gravity loading option	196
	Program 5.5 Plane or axisymmetric thermoelastic analysis of an elastic solid using 3-, 6-, 10- or 15-node right-angled triangles or 4-, 8- or 9-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction	205
	Program 5.6 Three-dimensional strain of a cuboidal elastic solid using 8-, 14- or 20-node brick hexahedra. Mesh numbered in $xz$ -planes then in the $y$ -direction. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver	209
	Program 5.7 Three-dimensional strain of a cuboidal elastic solid using 8-, 14- or 20-node brick hexahedra. Mesh numbered in $xz$ -planes then in the $y$ -direction. No global stiffness matrix. Diagonally preconditioned conjugate gradient solver. Optimised maths library, ABAQUS UMAT version	213
5.2	Glossary of Variable Names	221
5.3	Exercises	224
	References	232
<b>6</b>	<b>Material Non-linearity</b>	<b>233</b>
6.1	Introduction	233
6.2	Stress–strain Behaviour	235
6.3	Stress Invariants	236
6.4	Failure Criteria	238
	6.4.1 <i>Von Mises</i>	238
	6.4.2 <i>Mohr–Coulomb and Tresca</i>	239
6.5	Generation of Body Loads	240
6.6	Viscoplasticity	240

6.7	Initial Stress	242
6.8	Corners on the Failure and Potential Surfaces	243
	Program 6.1 Plane-strain-bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Flexible smooth footing. Load control. Viscoplastic strain method	244
	Program 6.2 Plane-strain-bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Flexible smooth footing. Load control. Viscoplastic strain method. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver	250
	Program 6.3 Plane-strain-bearing capacity analysis of an elastic–plastic (Mohr–Coulomb) material using 8-node rectangular quadrilaterals. Rigid smooth footing. Displacement control. Viscoplastic strain method	254
	Program 6.4 Plane-strain slope stability analysis of an elastic–plastic (Mohr–Coulomb) material using 8-node rectangular quadrilaterals. Gravity loading. Viscoplastic strain method	260
	Program 6.5 Plane-strain earth pressure analysis of an elastic–plastic (Mohr–Coulomb) material using 8-node rectangular quadrilaterals. Rigid smooth wall. Initial stress method	265
6.9	Elastoplastic Rate Integration	270
	6.9.1 Forward Euler Method	272
	6.9.2 Backward Euler Method	274
6.10	Tangent Stiffness Approaches	275
	6.10.1 Inconsistent Tangent Matrix	275
	6.10.2 Consistent Tangent Matrix	275
	6.10.3 Convergence Criterion	276
	Program 6.6 Plane-strain-bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Flexible smooth footing. Load control. Consistent tangent stiffness. Closest point projection method (CPPM)	277
	Program 6.7 Plane-strain-bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Flexible smooth footing. Load control. Consistent tangent stiffness. CPPM. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver	282
	Program 6.8 Plane-strain-bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Flexible smooth footing. Load control. Consistent tangent stiffness. Radial return method (RR) with ‘line search’	286
6.11	The Geotechnical Processes of Embanking and Excavation	289
	6.11.1 Embanking	289
	Program 6.9 Plane-strain construction of an elastic–plastic (Mohr–Coulomb) embankment in layers on a foundation using 8-node quadrilaterals. Viscoplastic strain method	290
	6.11.2 Excavation	294

	Program 6.10 Plane-strain construction of an elastic–plastic (Mohr–Coulomb) excavation in layers using 8-node quadrilaterals. Viscoplastic strain method	300
6.12	Undrained Analysis	305
	Program 6.11 Axisymmetric ‘undrained’ strain of an elastic–plastic (Mohr–Coulomb) solid using 8-node rectangular quadrilaterals. Viscoplastic strain method	308
	Program 6.12 Three-dimensional strain analysis of an elastic–plastic (Mohr–Coulomb) slope using 20-node hexahedra. Gravity loading. Viscoplastic strain method	313
	Program 6.13 Three-dimensional strain analysis of an elastic–plastic (Mohr–Coulomb) slope using 20-node hexahedra. Gravity loading. Viscoplastic strain method. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver	319
6.13	Glossary of Variable Names	322
6.14	Exercises	327
	References	331
<b>7</b>	<b>Steady State Flow</b>	<b>333</b>
7.1	Introduction	333
	Program 7.1 One-dimensional analysis of steady seepage using 2-node line elements	334
	Program 7.2 Plane or axisymmetric analysis of steady seepage using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction	337
	Program 7.3 Analysis of plane free surface flow using 4-node quadrilaterals. ‘Analytical’ form of element conductivity matrix	344
	Program 7.4 General two- (plane) or three-dimensional analysis of steady seepage	351
	Program 7.5 General two- (plane) or three-dimensional analysis of steady seepage. No global conductivity matrix assembly. Diagonally preconditioned conjugate gradient solver	355
7.2	Glossary of Variable Names	359
7.3	Exercises	361
	References	367
<b>8</b>	<b>Transient Problems: First Order (Uncoupled)</b>	<b>369</b>
8.1	Introduction	369
	Program 8.1 One-dimensional transient (consolidation) analysis using 2-node ‘line’ elements. Implicit time integration using the ‘theta’ method	370
	Program 8.2 One-dimensional transient (consolidation) analysis (settlement and excess pore pressure) using 2-node ‘line’ elements. Implicit time integration using the ‘theta’ method	373
	Program 8.3 One-dimensional consolidation analysis using 2-node ‘line’ elements. Explicit time integration. Element by element. Lumped mass	377

Program 8.4	Plane or axisymmetric transient (consolidation) analysis using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction. Implicit time integration using the ‘theta’ method	380
Program 8.5	Plane or axisymmetric transient (consolidation) analysis using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction. Implicit time integration using the ‘theta’ method. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver	388
Program 8.6	Plane or axisymmetric transient (consolidation) analysis using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction. Explicit time integration using the ‘theta = 0’ method	390
Program 8.7	Plane or axisymmetric transient (consolidation) analysis using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction. ‘theta’ method using an element-by-element product algorithm	394
8.2	Comparison of Programs 8.4, 8.5, 8.6 and 8.7	397
Program 8.8	General two- (plane) or three-dimensional transient (consolidation) analysis. Implicit time integration using the ‘theta’ method	397
Program 8.9	Plane analysis of the diffusion–convection equation using 4-node rectangular quadrilaterals. Implicit time integration using the ‘theta’ method. Self-adjoint transformation	401
Program 8.10	Plane analysis of the diffusion–convection equation using 4-node rectangular quadrilaterals. Implicit time integration using the ‘theta’ method. Untransformed solution	405
Program 8.11	Plane or axisymmetric transient thermal conduction analysis using 4-node rectangular quadrilaterals. Implicit time integration using the ‘theta’ method. Option of convection and flux boundary conditions	410
8.3	Glossary of Variable Names	416
8.4	Exercises	419
	References	422
<b>9</b>	<b>Coupled Problems</b>	<b>423</b>
9.1	Introduction	423
Program 9.1	Analysis of the plane steady-state Navier–Stokes equation using 8-node rectangular quadrilaterals for velocities coupled to 4-node rectangular quadrilaterals for pressures. Mesh numbered in $x$ -direction. Freedoms numbered in the order $u - p - v$	424
Program 9.2	Analysis of the plane steady-state Navier–Stokes equation using 8-node rectangular quadrilaterals for velocities coupled to 4-node rectangular quadrilaterals for pressures. Mesh numbered in $x$ -direction. Freedoms numbered in the order $u - p - v$ . Element-by-element solution using BiCGStab(l) with no preconditioning. No global matrix assembly	429

Program 9.3	One-dimensional coupled consolidation analysis of a Biot poroelastic solid using 2-node 'line' elements. Freedoms numbered in the order $v - u_w$	433
Program 9.4	Plane strain consolidation analysis of a Biot elastic solid using 8-node rectangular quadrilaterals for displacements coupled to 4-node rectangular quadrilaterals for pressures. Freedoms numbered in order $u - v - u_w$ . Incremental load version	438
Program 9.5	Plane strain consolidation analysis of a Biot elastic solid using 8-node rectangular quadrilaterals for displacements coupled to 4-node rectangular quadrilaterals for pressures. Freedoms numbered in order $u - v - u_w$ . Incremental load version. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver	445
Program 9.6	Plane strain consolidation analysis of a Biot poroelastic-plastic (Mohr-Coulomb) material using 8-node rectangular quadrilaterals for displacements coupled to 4-node rectangular quadrilaterals for pressures. Freedoms numbered in the order $u - v - u_w$ . Viscoplastic strain method	448
9.2	Glossary of Variable Names	454
9.3	Exercises	459
	References	460
<b>10</b>	<b>Eigenvalue Problems</b>	<b>461</b>
10.1	Introduction	461
Program 10.1	Eigenvalue analysis of elastic beams using 2-node beam elements. Lumped mass	462
Program 10.2	Eigenvalue analysis of an elastic solid in plane strain using 4- or 8-node rectangular quadrilaterals. Lumped mass. Mesh numbered in y-direction	465
Program 10.3	Eigenvalue analysis of an elastic solid in plane strain using 4-node rectangular quadrilaterals. Lanczos method. Consistent mass. Mesh numbered in y-direction	469
Program 10.4	Eigenvalue analysis of an elastic solid in plane strain using 4-node rectangular quadrilaterals with ARPACK. Lumped mass. Element-by-element formulation. Mesh numbered in y-direction	474
10.2	Glossary of Variable Names	477
10.3	Exercises	480
	References	482
<b>11</b>	<b>Forced Vibrations</b>	<b>483</b>
11.1	Introduction	483
Program 11.1	Forced vibration analysis of elastic beams using 2-node beam elements. Consistent mass. Newmark time stepping	483
Program 11.2	Forced vibration analysis of an elastic solid in plane strain using 4- or 8-node rectangular quadrilaterals. Lumped mass. Mesh numbered in the y-direction. Modal superposition	489

Program 11.3	Forced vibration analysis of an elastic solid in plane strain using rectangular 8-node quadrilaterals. Lumped or consistent mass. Mesh numbered in the y-direction. Implicit time integration using the ‘theta’ method	493
Program 11.4	Forced vibration analysis of an elastic solid in plane strain using rectangular 8-node quadrilaterals. Lumped or consistent mass. Mesh numbered in the y-direction. Implicit time integration using Wilson’s method	498
Program 11.5	Forced vibration of a rectangular elastic solid in plane strain using 8-node quadrilateral elements numbered in the y-direction. Lumped mass, complex response	501
Program 11.6	Forced vibration analysis of an elastic solid in plane strain using uniform size rectangular 4-node quadrilaterals. Mesh numbered in the y-direction. Lumped or consistent mass. Mixed explicit/ implicit time integration	504
Program 11.7	Forced vibration analysis of an elastic solid in plane strain using rectangular 8-node quadrilaterals. Lumped or consistent mass. Mesh numbered in the y-direction. Implicit time integration using the ‘theta’ method. No global matrix assembly. Diagonally preconditioned conjugate gradient solver	508
Program 11.8	Forced vibration analysis of an elastic–plastic (von Mises) solid in plane strain using rectangular 8-node quadrilateral elements. Lumped mass. Mesh numbered in the y-direction. Explicit time integration	512
11.2	Glossary of Variable Names	517
11.3	Exercises	521
	References	522
<b>12</b>	<b>Parallel Processing of Finite Element Analyses</b>	<b>523</b>
12.1	Introduction	523
12.2	Differences between Parallel and Serial Programs	525
12.2.1	<i>Parallel Libraries</i>	525
12.2.2	<i>Global Variables</i>	526
12.2.3	<i>MPI Library Routines</i>	526
12.2.4	<i>The _pp Appendage</i>	527
12.2.5	<i>Simple Test Problems</i>	527
12.2.6	<i>Reading and Writing</i>	530
12.2.7	<i>rest Instead of nf</i>	532
12.2.8	<i>Gathering and Scattering</i>	533
12.2.9	<i>Reindexing</i>	533
12.2.10	<i>Domain Composition</i>	533
12.2.11	<i>Third-party Mesh-partitioning Tools</i>	534
12.2.12	<i>Load Balancing</i>	535
Program 12.1	Three-dimensional analysis of an elastic solid. Compare Program 5.6	536
Program 12.2	Three-dimensional analysis of an elastoplastic (Mohr–Coulomb) solid. Compare Program 6.13	542
Program 12.3	Three-dimensional Laplacian flow. Compare Program 7.5	548

Program 12.4 Three-dimensional transient heat conduction–implicit analysis in time. Compare Program 8.5	553
Program 12.5 Three-dimensional transient flow–explicit analysis in time. Compare Program 8.6	562
Program 12.6 Three-dimensional steady-state Navier–Stokes analysis. Compare Program 9.2	565
Program 12.7 Three-dimensional analysis of Biot poro elastic solid. Incremental version. Compare Program 9.5	572
Program 12.8 Eigenvalue analysis of three-dimensional elastic solid. Compare Program 103	576
Program 12.9 Forced vibration analysis of a three-dimensional elastic solid. Implicit integration in time. Compare Program 11.7	581
Program 12.10 Forced vibration analysis of three-dimensional elastoplastic solid. Explicit integration in time. Compare Program 11.8	585
12.3 Graphics Processing Units	589
Program 12.11 Three-dimensional strain of an elastic solid using 8-, 14- or 20-node brick hexahedra. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver. GPU version. Compare Program 5.7	589
12.4 Cloud Computing	594
12.5 Conclusions	596
12.6 Glossary of Variable Names	597
References	602
<b>Appendix A Equivalent Nodal Loads</b>	<b>605</b>
<b>Appendix B Shape Functions and Element Node Numbering</b>	<b>611</b>
<b>Appendix C Plastic Stress-Strain Matrices and Plastic Potential Derivatives</b>	<b>619</b>
<b>Appendix D main Library Subprograms</b>	<b>623</b>
<b>Appendix E geom Library Subroutines</b>	<b>635</b>
<b>Appendix F Parallel Library Subroutines</b>	<b>639</b>
<b>Appendix G External Subprograms</b>	<b>645</b>
<b>Author Index</b>	<b>649</b>
<b>Subject Index</b>	<b>653</b>

# 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.

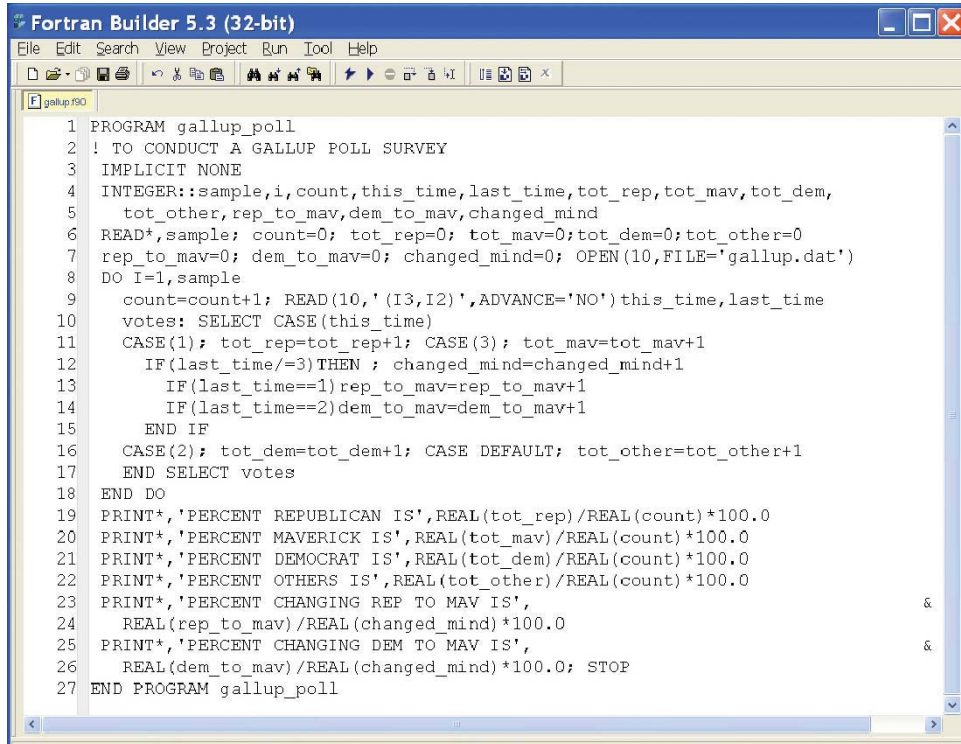
Figure 1.1 shows a typical simple program written in FORTRAN (Smith, 1995). It concerns an opinion poll survey and serves to illustrate the basic structure of the language for those used to other languages.

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



```

1 PROGRAM Gallup_poll
2 ! TO CONDUCT A GALLUP POLL SURVEY
3 IMPLICIT NONE
4 INTEGER::sample,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](http://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

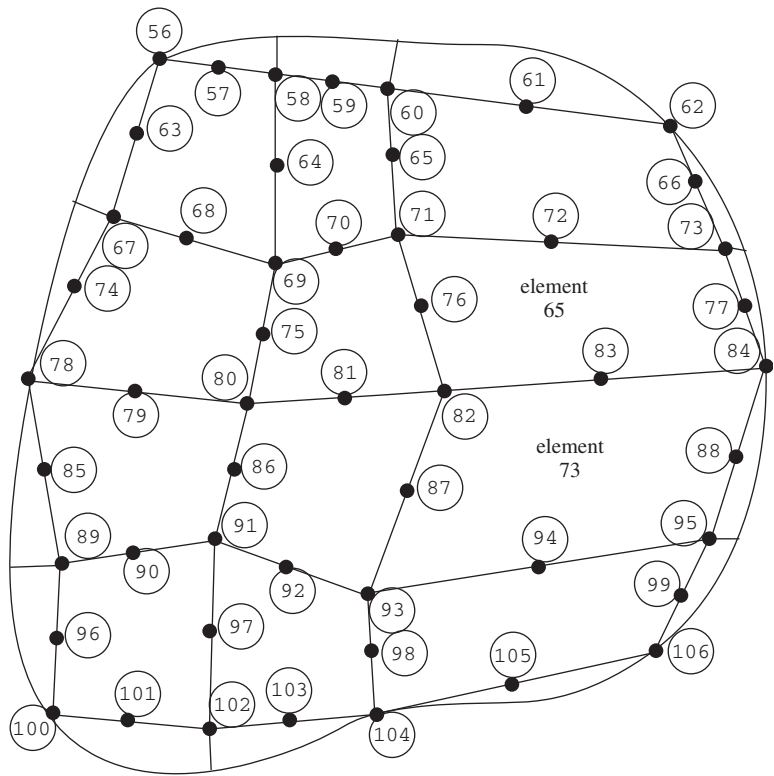


Figure 1.2 Portion of a finite element mesh with node and element numbers