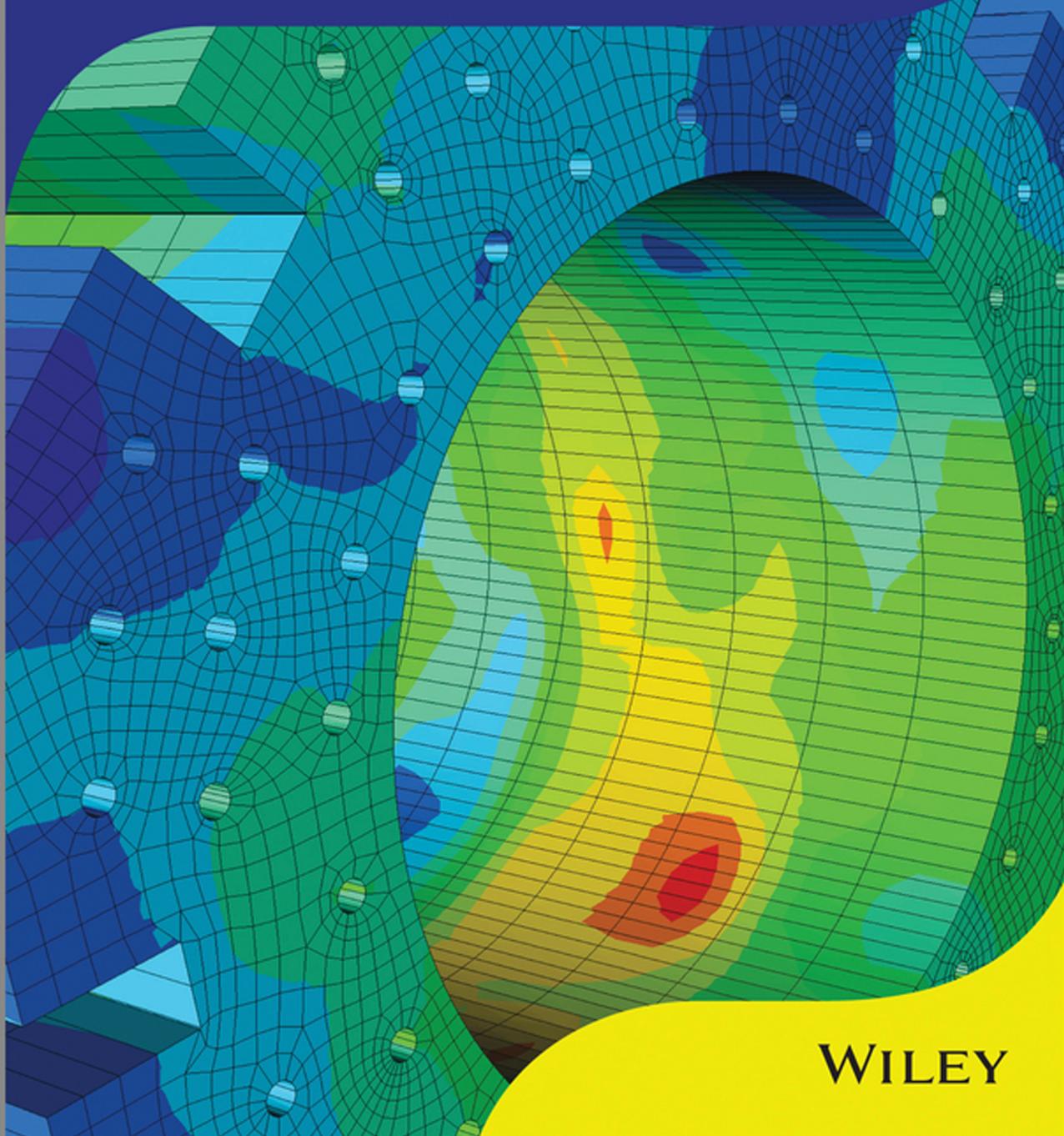


Fifth Edition

# 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.F5564 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	xv
Acknowledgements	xvii
<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

<i>1.11.3 Display Applied Loads</i>	21
<i>1.11.4 Display Deformed Mesh</i>	21
1.12 Conclusions	23
References	24
<b>2 Spatial Discretisation by Finite Elements</b>	<b>25</b>
2.1 Introduction	25
2.2 Rod Element	25
<i>2.2.1 Rod Stiffness Matrix</i>	25
<i>2.2.2 Rod Mass Element</i>	28
2.3 The Eigenvalue Equation	28
2.4 Beam Element	29
<i>2.4.1 Beam Element Stiffness Matrix</i>	29
<i>2.4.2 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
<i>2.10.1 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
<i>2.17.1 Steady State</i>	51
<i>2.17.2 Transient State</i>	53
<i>2.17.3 Convection</i>	53
2.18 Further Coupled Equations: Biot Consolidation	54
2.19 Conclusions	56
References	56
<b>3 Programming Finite Element Computations</b>	<b>59</b>
3.1 Introduction	59
3.2 Local Coordinates for Quadrilateral Elements	59
<i>3.2.1 Numerical Integration for Quadrilaterals</i>	61
<i>3.2.2 Analytical Integration for Quadrilaterals</i>	63
3.3 Local Coordinates for Triangular Elements	64
<i>3.3.1 Numerical Integration for Triangles</i>	65
<i>3.3.2 Analytical Integration for Triangles</i>	65
3.4 Multi-Element Assemblies	66
3.5 ‘Element-by-Element’ Techniques	68
<i>3.5.1 Conjugate Gradient Method for Linear Equation Systems</i>	68
<i>3.5.2 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	Incorporation of Boundary Conditions	72
3.6.1	<i>Convection Boundary Conditions</i>	74
3.7	Programming using Building Blocks	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	Solution of Equilibrium Equations	95
3.9	Evaluation of Eigenvalues and Eigenvectors	96
3.9.1	<i>Jacobi Algorithm</i>	96
3.9.2	<i>Lanczos and Arnoldi Algorithms</i>	98
3.10	Solution of First-Order Time-Dependent Problems	99
3.11	Solution of Coupled Navier–Stokes Problems	103
3.12	Solution of Coupled Transient Problems	104
3.12.1	<i>Absolute Load Version</i>	105
3.12.2	<i>Incremental Load Version</i>	106
3.13	Solution of Second-Order Time-Dependent Problems	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
	References	113
<b>4</b>	<b>Static Equilibrium of Structures</b>	<b>115</b>
4.1	Introduction	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 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 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 <i>Forward Euler Method</i>	272
	6.9.2 <i>Backward Euler Method</i>	274
6.10	Tangent Stiffness Approaches	275
	6.10.1 <i>Inconsistent Tangent Matrix</i>	275
	6.10.2 <i>Consistent Tangent Matrix</i>	275
	6.10.3 <i>Convergence Criterion</i>	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 <i>Embanking</i>	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 <i>Excavation</i>	294

Program 6.10 Plane-strain construction of an elastic–plastic (Mohr–Coulomb) excavation in layers using 8-node quadrilaterals. Viscoplastic strain method	300
<b>6.12 Undrained Analysis</b>	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
<b>6.13 Glossary of Variable Names</b>	322
<b>6.14 Exercises</b>	327
<b>References</b>	331
<b>7 Steady State Flow</b>	333
<b>7.1 Introduction</b>	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
<b>7.2 Glossary of Variable Names</b>	359
<b>7.3 Exercises</b>	361
<b>References</b>	367
<b>8 Transient Problems: First Order (Uncoupled)</b>	369
<b>8.1 Introduction</b>	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(I) 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
9.3 References	460
<b>10 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
10.3 References	482
<b>11 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 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 elasto plastic 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 <code>main</code> Library Subprograms</b>	<b>623</b>
<b>Appendix E <code>geom</code> 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

```

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

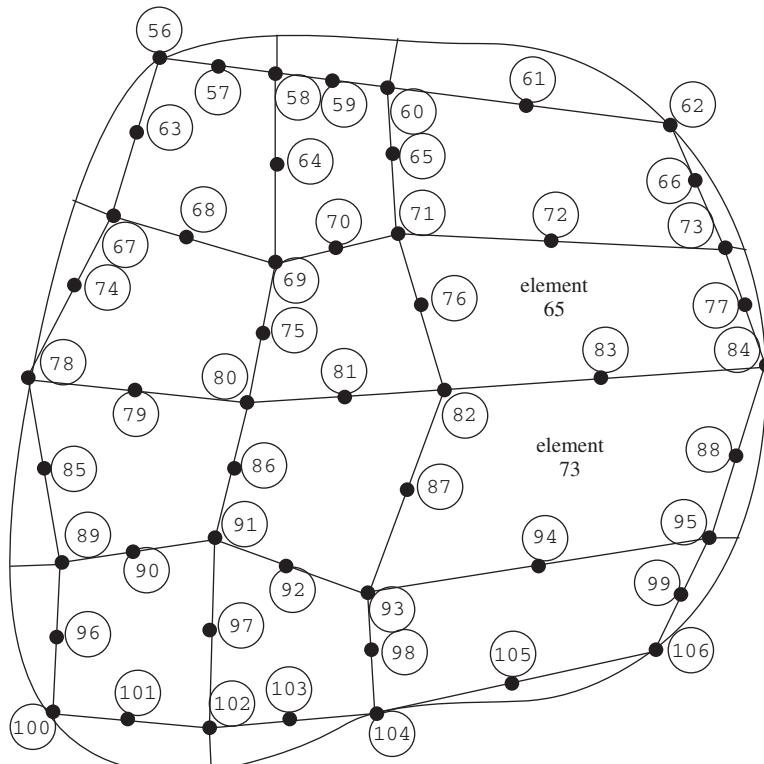
```
w = (/v, v/)
```

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

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.

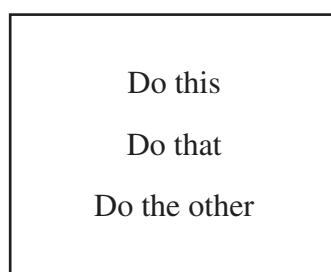
#### (ii) The choice

This corresponds to the IF...THEN...ELSE IF...THEN....END IF or SELECT CASE type of construct as shown in Figure 1.4.

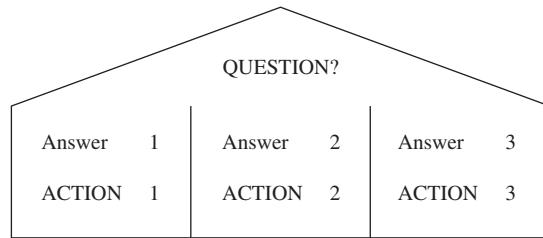
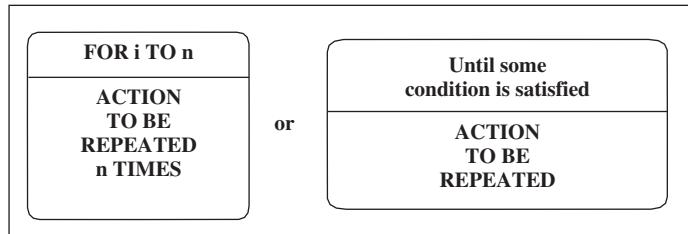
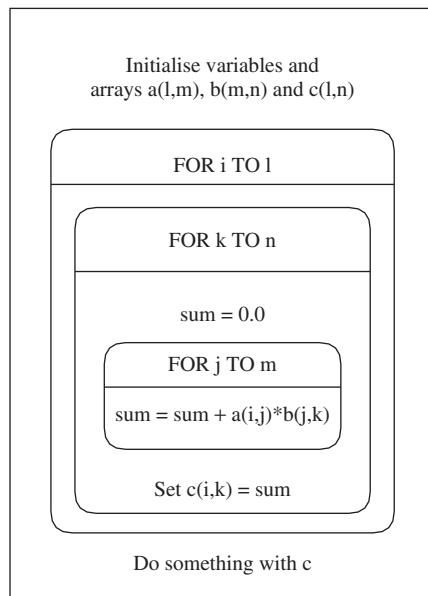
#### (iii) The loop

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](http://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](http://www.nag.co.uk)). The UK Rutherford Appleton Laboratory develops the Harwell Scientific Library ([www.hsl.rl.ac.uk](http://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](http://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](http://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 `nodef+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](http://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](http://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 Ensight Gold format:

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

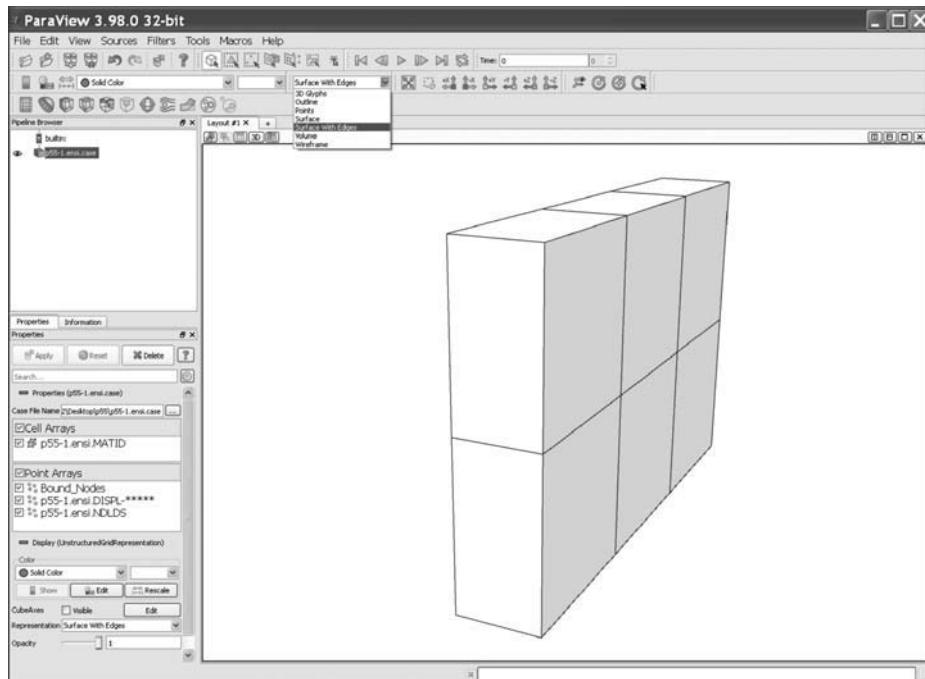
For Program 5.6, say, a typical set of output files generated by these two subroutines will contain

```
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.

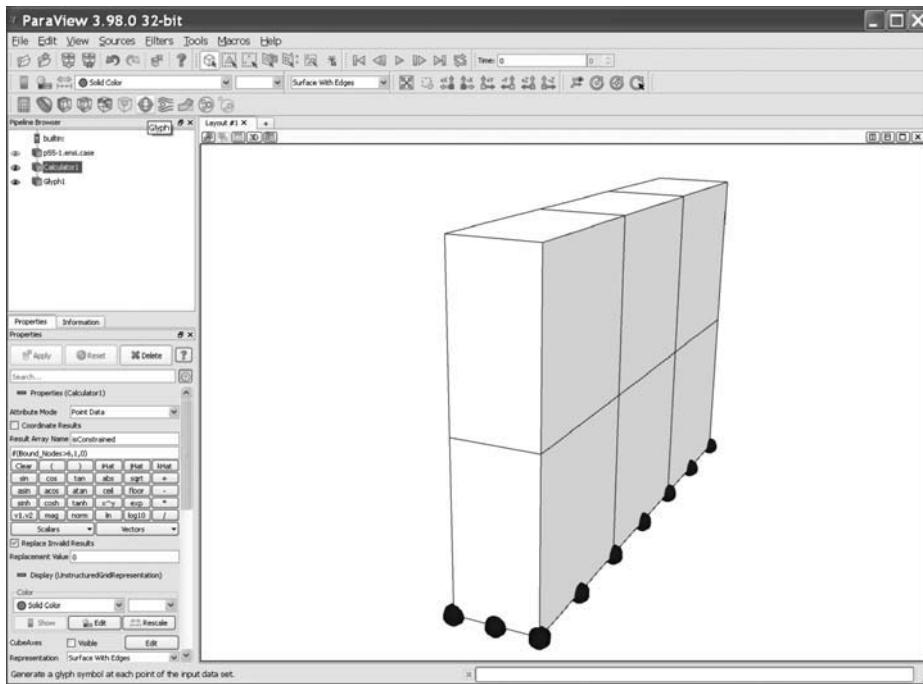
### 1.11.2 Display Restrained Nodes

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.



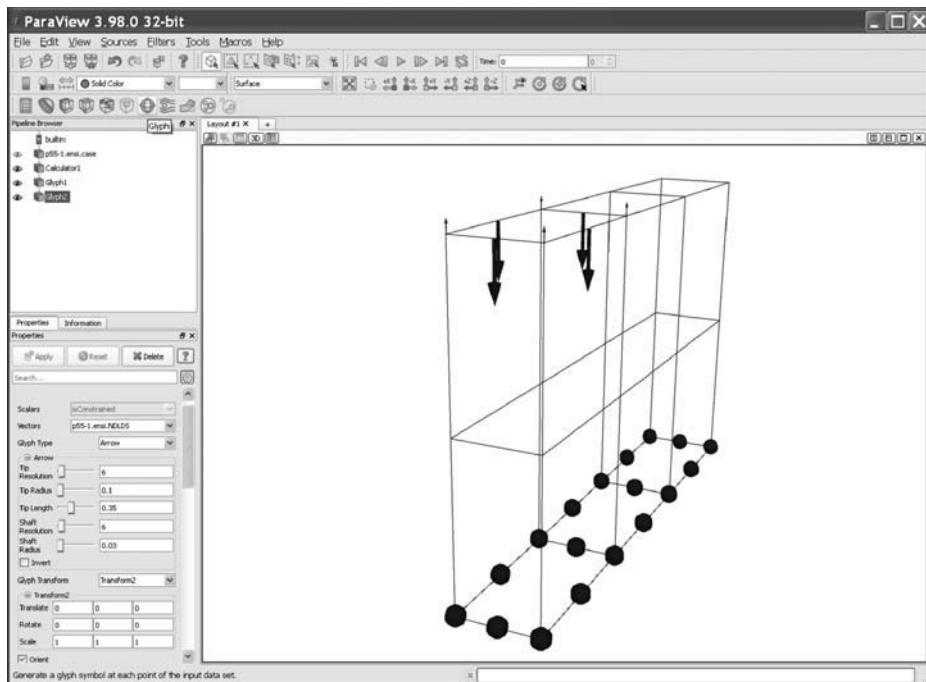
**Figure 1.8** Restrained nodes

### 1.11.3 Display Applied Loads

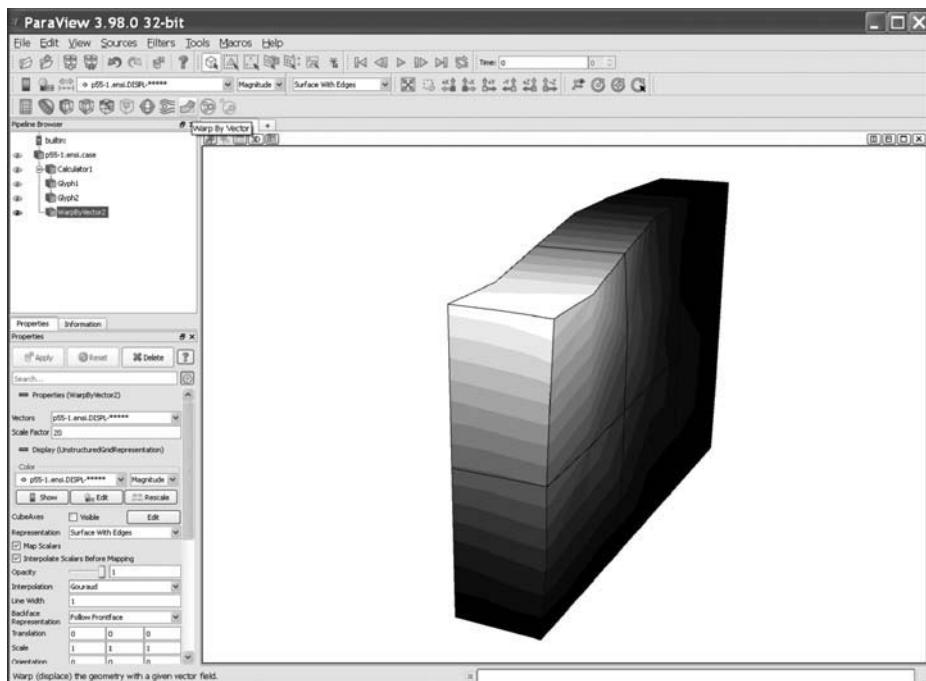
To view the applied loads, as shown in Figure 1.9, click on the p56.ensi.case file in the Pipeline Browser and then select a glyph filter using the Glyph icon. In the Properties tab, change the Scale Mode to Vector, set Vectors to load, select Glyph Type as Arrow and press Apply. Some arrows should appear 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](http://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.

# 2

## Spatial Discretisation by Finite Elements

### 2.1 Introduction

The finite element method is a technique for solving partial differential equations by first discretising these equations in their space dimensions. The discretisation is carried out locally over small regions of simple but arbitrary shape (the finite elements). This results in matrix equations relating the input at specified points in the elements (the nodes) to the output at these same points. In order to solve equations over large regions, the matrix equations for the smaller subregions can be summed node by node, resulting in global matrix equations, or ‘element-by-element’ techniques can be employed to avoid creating (large) global matrices. The method is already described in many texts, for example Zienkiewicz *et al.* (2005), Strang and Fix (2008), Cook *et al.* (2002) and Rao (2010), but the principles will briefly be described in this chapter in order to establish a notation and to set the scene for the later descriptions of programming techniques.

### 2.2 Rod Element

#### 2.2.1 Rod Stiffness Matrix

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

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

$$P = \sigma A = EA\epsilon = EA \frac{du}{dx} \quad (2.1)$$

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

$$\frac{dP}{dx} + F = 0 \quad (2.2)$$

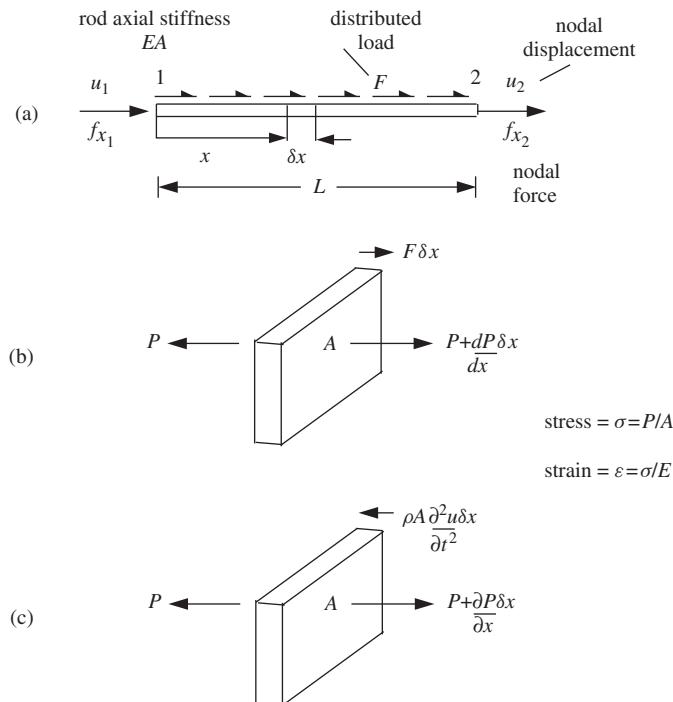


Figure 2.1 Equilibrium of a rod element

Hence the differential equation to be solved is

$$EA \frac{d^2u}{dx^2} + F = 0 \quad (2.3)$$

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

$$\tilde{u} = N_1 u_1 + N_2 u_2$$

or

$$\tilde{u} = [N_1 \ N_2] \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = [\mathbf{N}] \{\mathbf{u}\} \quad (2.4)$$

where

$$N_1 = 1 - \frac{x}{L}, \quad N_2 = \frac{x}{L} \quad (2.5)$$

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

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

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

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

Many methods could be used to achieve this. For example, Griffiths and Smith (2006) discuss collocation, subdomain, Galerkin and least squares techniques. Of these, Galerkin’s method, e.g. Finlayson (1972), is the most widely used in finite element work. The method consists of multiplying or ‘weighting’ the residual in (2.6) by each shape function in turn, integrating over the element and equating to zero. Thus

$$\int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} EA \frac{d^2}{dx^2} [N_1 \ N_2] dx \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} + \int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} F dx = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \quad (2.7)$$

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

$$\int N_i \frac{d^2 N_j}{dx^2} dx = - \int \frac{dN_i}{dx} \frac{dN_j}{dx} dx + \text{boundary terms which we usually ignore} \quad (2.8)$$

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

$$-EA \int_0^L \begin{bmatrix} \frac{dN_1}{dx} \frac{dN_1}{dx} & \frac{dN_1}{dx} \frac{dN_2}{dx} \\ \frac{dN_2}{dx} \frac{dN_1}{dx} & \frac{dN_2}{dx} \frac{dN_2}{dx} \end{bmatrix} dx \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} + F \int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} dx = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \quad (2.9)$$

On evaluation of the integrals,

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

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

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

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

$$[\mathbf{k}_m]\{\mathbf{u}\} = \{\mathbf{f}\} \quad (2.12)$$

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

### 2.2.2 Rod Mass Element

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

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

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

$$- \int_0^L \begin{Bmatrix} N_1 \\ N_2 \end{Bmatrix} \rho A [N_1 \ N_2] dx \frac{d^2}{dt^2} \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} \quad (2.14)$$

and assuming that  $\rho A$  is not a function of  $x$ ,

$$- \rho A \int_0^L \begin{bmatrix} N_1 N_1 & N_1 N_2 \\ N_2 N_1 & N_2 N_2 \end{bmatrix} dx \frac{d^2}{dt^2} \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} \quad (2.15)$$

Evaluation of integrals yields

$$- \frac{\rho A L}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \frac{d^2}{dt^2} \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} \quad (2.16)$$

or in matrix notation

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

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

$$[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{m}_m] \left\{ \frac{d^2 \mathbf{u}}{dt^2} \right\} = \{\mathbf{0}\} \quad (2.17)$$

which is a set of ordinary differential equations.

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

## 2.3 The Eigenvalue Equation

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

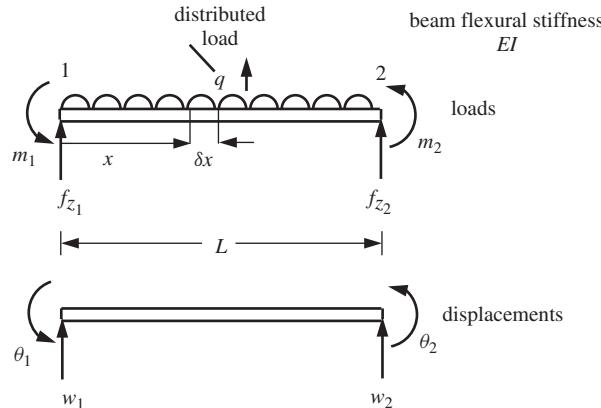
Suppose the elastic rod element is undergoing free harmonic motion. Then all nodal displacements will be harmonic, of the form

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

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

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

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



**Figure 2.2** Slender beam element

## 2.4 Beam Element

### 2.4.1 Beam Element Stiffness Matrix

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

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

$$EI \frac{d^4w}{dx^4} = q \quad (2.20)$$

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

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

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

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

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

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

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

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

$$\int N_i \frac{d^4 N_j}{dx^4} dx \approx - \int \frac{dN_i}{dx} \frac{d^3 N_j}{dx^3} dx \approx \int \frac{d^2 N_i}{dx^2} \frac{d^2 N_j}{dx^2} dx + \text{neglected terms} \quad (2.24)$$

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

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

Evaluation of the integrals gives

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

which recovers the standard ‘slope–deflection’ equations for beam elements.

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

$$\frac{2EI}{L^3} \begin{bmatrix} 6 & 3L & -6 & 3L \\ & 2L^2 & -3L & L^2 \\ & & 6 & -3L \\ & \text{symmetrical} & & 2L^2 \end{bmatrix} \begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} = \begin{Bmatrix} f_{z1} \\ m_1 \\ f_{z2} \\ m_2 \end{Bmatrix} \quad (2.27)$$

which represents the beam element stiffness relationship.

Hence, in matrix notation we again have

$$[\mathbf{k}_m]\{\mathbf{w}\} = \{\mathbf{f}\} \quad (2.28)$$

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

### 2.4.2 Beam Element Mass Matrix

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

$$-\rho A \int_0^L \begin{bmatrix} N_1 N_1 & N_1 N_2 & N_1 N_3 & N_1 N_4 \\ N_2 N_1 & N_2 N_2 & N_2 N_3 & N_2 N_4 \\ N_3 N_1 & N_3 N_2 & N_3 N_3 & N_3 N_4 \\ N_4 N_1 & N_4 N_2 & N_4 N_3 & N_4 N_4 \end{bmatrix} dx \frac{d^2}{dt^2} \begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} \quad (2.29)$$

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

$$[\mathbf{m}_m] = \frac{\rho A L}{420} \begin{bmatrix} 156 & 22L & 54 & -13L \\ & 4L^2 & 13L & -3L^2 \\ & & 156 & -22L \\ & & & 4L^2 \end{bmatrix} \quad (2.30)$$

symmetrical

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

## 2.5 Beam with an Axial Force

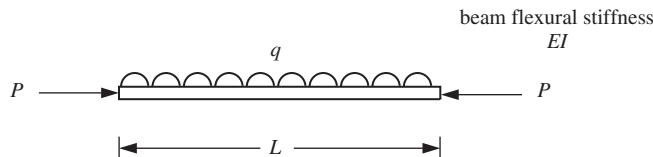
If the beam element in Figure 2.2 is subjected to an additional axial force  $P$  as shown in Figure 2.3, a simple modification to (2.20) results in the differential equation

$$EI \frac{d^4 w}{dx^4} \pm P \frac{d^2 w}{dx^2} = q \quad (2.31)$$

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

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

$$\mp P \int_0^L \left[ \frac{dN_i}{dx} \frac{dN_j}{dx} \right]_{i,j=1,2,3,4} dx \begin{Bmatrix} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{Bmatrix} \quad (2.32)$$



**Figure 2.3** Beam with an axial force

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

$$P \frac{1}{30L} \begin{bmatrix} 36 & 3L & -36 & 3L \\ & 4L^2 & -3L & -L^2 \\ & & 36 & -3L \\ & & & 4L^2 \end{bmatrix} \begin{cases} w_1 \\ \theta_1 \\ w_2 \\ \theta_2 \end{cases} \quad (2.33)$$

symmetrical

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

$$[\mathbf{g}_m] = \frac{1}{30L} \begin{bmatrix} 36 & 3L & -36 & 3L \\ & 4L^2 & -3L & -L^2 \\ & & 36 & -3L \\ & & & 4L^2 \end{bmatrix} \quad (2.34)$$

symmetrical

and the equilibrium equation can be written as

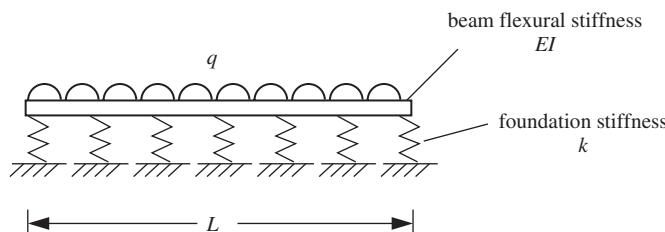
$$([\mathbf{k}_m] - P[\mathbf{g}_m])\{\mathbf{w}\} = \{\mathbf{f}\} \quad (2.35)$$

Buckling of a member can be investigated by solving the eigenvalue problem where  $\{\mathbf{f}\} = \{\mathbf{0}\}$ , or by increasing the compressive force  $P$  on the element until large deformations result or in simple cases, by determinant search. Equations (2.34) and (2.35) represent an approximation of the approach to modifying the element stiffness involving stability functions (e.g., Horne and Merchant, 1965). The accuracy of the approximation depends on the value of  $P/P_E$  for each member, where  $P_E$  is the Euler load. Over the range  $-1 < P/P_E < 1$  the approximation introduces errors no greater than 7% (Livesley, 1975). For larger positive values of  $P/P_E$ , however, (2.35) can become inaccurate unless more element subdivisions are used. Program 4.6 uses a simple iterative approach to compute the lowest buckling load of beams and beams on elastic foundations.

## 2.6 Beam on an Elastic Foundation

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

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



**Figure 2.4** Beam on a continuous elastic foundation

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

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

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

## 2.7 General Remarks on the Discretisation Process

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

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

## 2.8 Alternative Derivation of Element Stiffness

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

$$\delta U = \frac{1}{2} \frac{M^2}{EI} \delta x \quad (2.38)$$

**Table 2.1** Semi-discretisation of partial differential equations

Term in differential equation	Typical term in matrix equation	Symmetry?
$u$	$\int N_i N_j dx$	Yes
$\frac{du}{dx}$	$\int N_i \frac{dN_j}{dx} dx$	No
$\frac{d^2 u}{dx^2}$	$-\int \frac{dN_i}{dx} \frac{dN_j}{dx} dx$	Yes
$\frac{d^4 u}{dx^4}$	$\int \frac{d^2 N_i}{dx^2} \frac{d^2 N_j}{dx^2} dx$	Yes

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

$$\delta W = \frac{1}{2} q w \delta x \quad (2.39)$$

The bending moment  $M$  is related to  $w$  through the ‘moment–curvature’ expression

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

or

$$M = [\mathbf{D}]\{\mathbf{A}\}w \quad (2.40)$$

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

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

we have

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

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

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

The total strain energy of the element is thus

$$U = \frac{1}{2} \int_0^L \{\mathbf{w}\}^T (\{\mathbf{A}\}[\mathbf{N}])^T [\mathbf{D}]\{\mathbf{A}\}[\mathbf{N}]\{\mathbf{w}\} dx \quad (2.44)$$

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

$$U = \frac{1}{2} \{\mathbf{w}\}^T \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx \{\mathbf{w}\} \quad (2.45)$$

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

$$\begin{aligned} \Pi &= U - W \\ &= \frac{1}{2} \{\mathbf{w}\}^T \int_0^L [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx \{\mathbf{w}\} - \frac{1}{2} \{\mathbf{w}\}^T q \int_0^L [\mathbf{N}]^T dx \end{aligned} \quad (2.46)$$

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

$$\frac{\partial \Pi}{\partial \{\mathbf{w}\}^T} = \int_0^L [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx \{\mathbf{w}\} - q \int_0^L [\mathbf{N}]^T dx = 0 \quad (2.47)$$

or

$$\int_0^L [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx \{\mathbf{w}\} = q \int_0^L [\mathbf{N}]^T dx \quad (2.48)$$

which is simply another way of writing (2.25).

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

$$[\mathbf{k}_m] = \int_0^L [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx \quad (2.49)$$

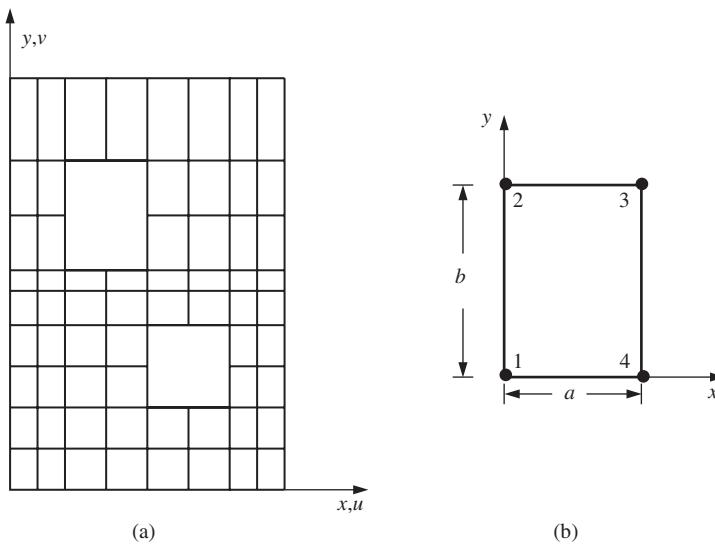
which will prove to be a useful general matrix form for expressing stiffnesses of all elastic solid elements. The computer programs for analysis of solids developed in the next chapter use this notation and method of stiffness formation.

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

## 2.9 Two-dimensional Elements: Plane Stress

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

This situation changes radically when problems in two or three space dimensions are analysed. For example, consider the plane shear wall with openings shown in Figure 2.5(a).



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

The wall has been subdivided into rectangular elements of side lengths  $a$  and  $b$ , of which Figure 2.5(b) is typical. These elements have four corner nodes so that when the idealised wall is assembled, the elements will only be attached at these points.

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

### 1. Equilibrium

$$\begin{aligned}\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + F_x &= 0 \\ \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + F_y &= 0\end{aligned}\quad (2.50)$$

where  $\sigma_x$ ,  $\sigma_y$  and  $\tau_{xy}$  are the only non-zero stress components and  $F_x$ ,  $F_y$  are ‘body forces’ (units of force/length<sup>3</sup>).

### 2. Constitutive (plane stress)

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

where  $E$  is Young’s modulus,  $\nu$  is Poisson’s ratio and  $\epsilon_x$ ,  $\epsilon_y$  and  $\gamma_{xy}$  are the independent small strain components.

### 3. Strain-displacement

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

where  $u$  and  $v$  are the components of displacement in the  $x$ - and  $y$ -directions. Equations (2.50)–(2.52) can be written in the form

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

where

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

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

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

$$\begin{aligned} [\mathbf{A}]^T \{\sigma\} &= -\{\mathbf{f}\} \\ [\mathbf{A}]^T [\mathbf{D}] \{\epsilon\} &= -\{\mathbf{f}\} \\ [\mathbf{A}]^T [\mathbf{D}] [\mathbf{A}] \{e\} &= -\{\mathbf{f}\} \end{aligned} \quad (2.56)$$

Writing out (2.56) in full we have

$$\frac{E}{1-\nu^2} \begin{Bmatrix} \frac{\partial^2 u}{\partial x^2} + \frac{1-\nu}{2} \frac{\partial^2 u}{\partial y^2} + \nu \frac{\partial^2 v}{\partial x \partial y} + \frac{1-\nu}{2} \frac{\partial^2 v}{\partial y \partial x} \\ \frac{\partial^2 v}{\partial y^2} + \frac{1-\nu}{2} \frac{\partial^2 v}{\partial x^2} + \nu \frac{\partial^2 u}{\partial y \partial x} + \frac{1-\nu}{2} \frac{\partial^2 u}{\partial x \partial y} \end{Bmatrix} = \begin{Bmatrix} -F_x \\ -F_y \end{Bmatrix} \quad (2.57)$$

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

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

$$\tilde{u} = [N_1 \ N_2 \ N_3 \ N_4] \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix} = [\mathbf{N}] \{\mathbf{u}\} \quad (2.58)$$

and

$$\tilde{v} = [N_1 \ N_2 \ N_3 \ N_4] \begin{Bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{Bmatrix} = [\mathbf{N}] \{\mathbf{v}\} \quad (2.59)$$

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

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

These result in linear variations in strain across the element, which is sometimes called the ‘linear strain rectangle’.

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

$$\frac{E}{1-\nu^2} \int_0^a \int_0^b \left[ \begin{array}{c} \left( \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} + \frac{1-\nu}{2} \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial y} \right) \\ \times \left( \nu \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial y} + \frac{1-\nu}{2} \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial x} \right) \\ \left( \nu \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial x} + \frac{1-\nu}{2} \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial y} \right) \\ \times \left( \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial y} + \frac{1-\nu}{2} \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} \right) \end{array} \right]_{i,j=1,2,3,4} dx dy \{u\} = \{f\} \quad (2.61)$$

where  $\{u\}$  and  $\{f\}$  are the nodal displacements and force components. Most programs in this book arrange these components by alternating them, thus  $\{u\} = [u_1 \ v_1 \ u_2 \ v_2 \ u_3 \ v_3 \ u_4 \ v_4]^T$  and  $\{f\} = [f_{x1} \ f_{y1} \ f_{x2} \ f_{y2} \ f_{x3} \ f_{y3} \ f_{x4} \ f_{y4}]^T$  where  $u_1$  is the  $x$ -displacement at node 1, and  $f_{y2}$  is the  $y$ -force at node 2, etc.

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

$$[k_m]\{u\} = \{f\} \quad (2.62)$$

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

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

and so on.

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

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

$$\int \int N_i \frac{\partial^2 N_j}{\partial x^2} dx dy = - \int \int \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} dx dy + \int_S N_i \frac{\partial N_j}{\partial x} l_n dS \quad (2.64)$$

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

## 2.10 Energy Approach and Plane Strain

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

The element strain energy per unit thickness is

$$\begin{aligned} U &= \int \int \frac{1}{2} \{\sigma\}^T \{\epsilon\} dx dy \\ &= \frac{1}{2} \{\mathbf{u}\}^T \int \int ([\mathbf{A}][\mathbf{S}])^T \mathbf{D}([\mathbf{A}][\mathbf{S}]) dx dy \{\mathbf{u}\} \end{aligned} \quad (2.65)$$

$$= \frac{1}{2} \{\mathbf{u}\}^T \int \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy \{\mathbf{u}\} \quad (2.66)$$

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

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

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

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

Thus we have again for this element

$$[\mathbf{k}_m] = \int \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy \quad (2.69)$$

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

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

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

### 2.10.1 Thermoelasticity

When a material is subjected to a temperature change given by  $\Delta T$ , it tries to expand (or contract). If free expansion is inhibited, for example due to boundary conditions, thermal stresses are generated. If we assume a 2D anisotropic material with thermal expansion coefficients  $\alpha_x$  and  $\alpha_y$ , then the thermal strains are given by

$$\{\boldsymbol{\epsilon}^t\} = \begin{Bmatrix} \epsilon_x^t \\ \epsilon_y^t \\ \gamma_{xy}^t \end{Bmatrix} = \begin{Bmatrix} \alpha_x \Delta T \\ \alpha_y \Delta T \\ 0 \end{Bmatrix} \quad (2.71)$$

The nodal forces on an element that would cause these strains to occur in an elastic material are given by

$$\{f^t\} = \int \int [\mathbf{B}]^T [\mathbf{D}] \{\epsilon^t\} dx dy \quad (2.72)$$

which may be assembled for all thermally affected elements into a global thermal loads vector. The thermal loads may then be added to any additional loading required, after which a conventional finite element formulation leads to the global nodal displacements of the system.

In order to retrieve stresses in a thermally loaded system, the thermal strains must be subtracted from the total strains before multiplying by the constitutive matrix. Thus in an elastic material, returning to the element level with nodal displacements  $\{\mathbf{u}\}$ , we get

$$\{\epsilon\} = [\mathbf{B}]\{\mathbf{u}\} \quad (2.73)$$

$$\{\sigma\} = [\mathbf{D}](\{\epsilon\} - \{\epsilon^t\}) \quad (2.74)$$

## 2.11 Plane Element Mass Matrix

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

$$[\mathbf{m}_m] = \rho \int \int [\mathbf{N}]^T [\mathbf{N}] dx dy \quad (2.75)$$

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

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

$$m_{m1,1} = \frac{\rho ab}{9} \quad (2.76)$$

## 2.12 Axisymmetric Stress and Strain

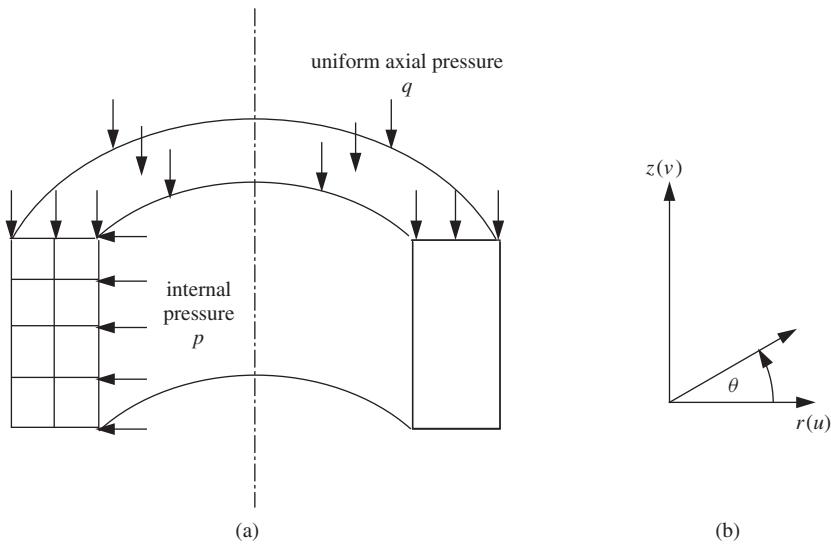
Solids of revolution subjected to axisymmetric loading possess only two independent components of displacement and can be analysed as if they were two-dimensional. For example, Figure 2.6(a) shows a thick tube subjected to radial pressure  $p$  and axial pressure  $q$ .

Only a typical radial cross-section need be analysed and is subdivided into rectangular elements in the figure. The cylindrical coordinate system, Figure 2.6(b), is the most convenient and when it is used the element stiffness equation equivalent to (2.69) is

$$[\mathbf{k}_m] = \int \int \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] r dr dz d\theta \quad (2.77)$$

which, when integrated over one radian, becomes

$$[\mathbf{k}_m] = \int \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] r dr dz \quad (2.78)$$



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

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

$$\begin{Bmatrix} \epsilon_r \\ \epsilon_z \\ \gamma_{rz} \\ \epsilon_\theta \end{Bmatrix} = \begin{bmatrix} \frac{\partial}{\partial r} & 0 \\ 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial z} & \frac{\partial}{\partial r} \\ \frac{1}{r} & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix} \quad (2.79)$$

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

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

$$[\mathbf{B}] = \begin{bmatrix} \frac{\partial N_1}{\partial r} & 0 & \frac{\partial N_2}{\partial r} & 0 & \frac{\partial N_3}{\partial r} & 0 & \frac{\partial N_4}{\partial r} & 0 \\ 0 & \frac{\partial N_1}{\partial z} & 0 & \frac{\partial N_2}{\partial z} & 0 & \frac{\partial N_3}{\partial z} & 0 & \frac{\partial N_4}{\partial z} \\ \frac{\partial N_1}{\partial z} & \frac{\partial N_1}{\partial r} & \frac{\partial N_2}{\partial z} & \frac{\partial N_2}{\partial r} & \frac{\partial N_3}{\partial z} & \frac{\partial N_3}{\partial r} & \frac{\partial N_4}{\partial z} & \frac{\partial N_4}{\partial r} \\ \frac{N_1}{r} & 0 & \frac{N_2}{r} & 0 & \frac{N_3}{r} & 0 & \frac{N_4}{r} & 0 \end{bmatrix} \quad (2.80)$$

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

$$[\mathbf{D}] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & 1 & 0 & \frac{\nu}{1-\nu} \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 1 \end{bmatrix} \quad (2.81)$$

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

## 2.13 Three-dimensional Stress and Strain

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

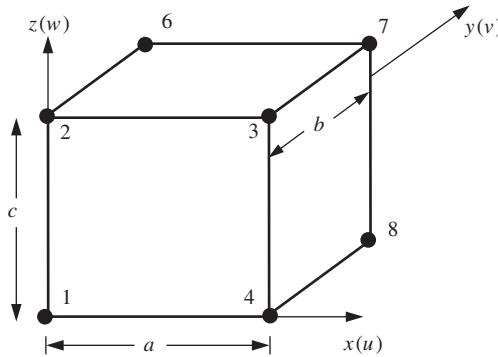
$$[\mathbf{k}_m] = \int \int \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy dz \quad (2.82)$$

where the full strain–displacement relations are (Timoshenko and Goodier, 1982)

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{Bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \quad (2.83)$$

or

$$\{\boldsymbol{\epsilon}\} = [\mathbf{A}]\{\mathbf{e}\} \quad (2.84)$$



**Figure 2.7** 8-node ‘brick element’

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

$$[\mathbf{N}] = [N_1 \ N_2 \ N_3 \ N_4 \ N_5 \ N_6 \ N_7 \ N_8] \quad (2.85)$$

where

$$\begin{aligned} N_1 &= \left(1 - \frac{x}{a}\right) \left(1 - \frac{y}{b}\right) \left(1 - \frac{z}{c}\right) \\ N_2 &= \left(1 - \frac{x}{a}\right) \left(1 - \frac{y}{b}\right) \frac{z}{c} \\ N_3 &= \frac{x}{a} \left(1 - \frac{y}{b}\right) \frac{z}{c} \\ N_4 &= \frac{x}{a} \left(1 - \frac{y}{b}\right) \left(1 - \frac{z}{c}\right) \quad (2.86) \\ N_5 &= \left(1 - \frac{x}{a}\right) \frac{y}{b} \left(1 - \frac{z}{c}\right) \\ N_6 &= \left(1 - \frac{x}{a}\right) \frac{y}{b} \frac{z}{c} \\ N_7 &= \frac{x}{a} \frac{y}{b} \frac{z}{c} \\ N_8 &= \frac{x}{a} \frac{y}{b} \left(1 - \frac{z}{c}\right) \end{aligned}$$

The full  $[\mathbf{S}]$  matrix would be of the form

$$[\mathbf{S}] = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & \dots \\ 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & \dots \\ 0 & 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & \dots \end{bmatrix} \quad (2.87)$$

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

$$[\mathbf{D}] = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \quad (2.88)$$

## 2.14 Plate Bending Element

The bending of a thin plate is governed by the equation

$$D\nabla^4 w = q \quad (2.89)$$

where  $\nabla^4$  is the biharmonic operator,  $D$  is the flexural rigidity of the plate, given by

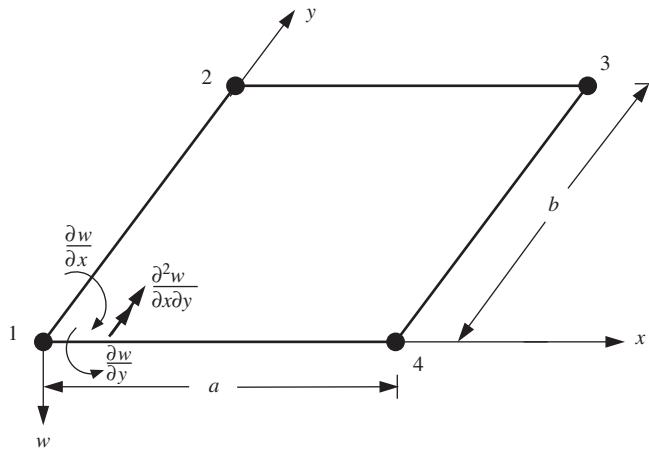
$$D = \frac{Eh^3}{12(1-\nu^2)} \quad (2.90)$$

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

Solution of (2.89) directly, for example by Galerkin's method, appears to imply that for a fixed  $D$ , a thin plate's deflection is unaffected by the value of Poisson's ratio. This is in fact only true for certain boundary conditions and in general the integration by parts in the Galerkin process will supply extra terms that are dependent on  $\nu$ .

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

$$U = \frac{1}{2}D \int \int \left\{ \left( \frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} \right)^2 - 2(1-\nu) \left[ \frac{\partial^2 w}{\partial x^2} \frac{\partial^2 w}{\partial y^2} - \left( \frac{\partial^2 w}{\partial x \partial y} \right)^2 \right] \right\} dx dy \quad (2.91)$$



**Figure 2.8** Rectangular plate bending element

or

$$U = \frac{1}{2}D \int \int \left\{ \left( \frac{\partial^2 w}{\partial x^2} \right)^2 + \left( \frac{\partial^2 w}{\partial y^2} \right)^2 + 2\nu \frac{\partial^2 w}{\partial x^2} \frac{\partial^2 w}{\partial y^2} + 2(1-\nu) \left( \frac{\partial^2 w}{\partial x \partial y} \right)^2 \right\} dx dy \quad (2.92)$$

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

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

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

$$\tilde{w} = [\mathbf{N}]\{\mathbf{w}\} \quad (2.93)$$

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

$$N_1 = \frac{1}{a^3}(a^3 - 3ax^2 + 2x^3) \frac{1}{b^3}(b^3 - 3by^2 + 2y^3) \quad (2.94)$$

Defining

$$P_1 = \frac{1}{a^3}(a^3 - 3ax^2 + 2x^3)$$

$$Q_1 = \frac{1}{b^3}(b^3 - 3by^2 + 2y^3)$$

$$\begin{aligned}
P_2 &= \frac{1}{a^2}(a^2x - 2ax^2 + x^3) \\
Q_2 &= \frac{1}{b^2}(b^2y - 2by^2 + y^3) \\
P_3 &= \frac{1}{a^3}(3ax^2 - 2x^3) \\
Q_3 &= \frac{1}{b^3}(3by^2 - 2y^3) \\
P_4 &= \frac{1}{a^2}(x^3 - ax^2) \\
Q_4 &= \frac{1}{b^2}(y^3 - by^2)
\end{aligned} \tag{2.95}$$

the list of shape functions becomes

$$\begin{aligned}
N_1 &= P_1 Q_1 & N_9 &= P_3 Q_3 \\
N_2 &= P_2 Q_1 & N_{10} &= P_4 Q_3 \\
N_3 &= P_1 Q_2 & N_{11} &= P_3 Q_4 \\
N_4 &= P_2 Q_2 & N_{12} &= P_4 Q_4 \\
N_5 &= P_1 Q_3 & N_{13} &= P_3 Q_1 \\
N_6 &= P_2 Q_3 & N_{14} &= P_4 Q_1 \\
N_7 &= P_1 Q_4 & N_{15} &= P_3 Q_2 \\
N_8 &= P_2 Q_4 & N_{16} &= P_4 Q_2
\end{aligned} \tag{2.96}$$

Using the same energy formulation as before, and defining

$$\begin{Bmatrix} M_x \\ M_y \\ M_{xy} \end{Bmatrix} = D \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \begin{Bmatrix} \frac{\partial^2}{\partial x^2} \\ \frac{\partial^2}{\partial y^2} \\ 2 \frac{\partial^2}{\partial x \partial y} \end{Bmatrix} w \tag{2.97}$$

or

$$\{\mathbf{M}\} = [\mathbf{D}]\{\mathbf{A}\}w \tag{2.98}$$

it can readily be verified that (2.92) for strain energy can be written

$$U = \frac{1}{2} \int \int (\{\mathbf{A}\}w)^T [\mathbf{D}] (\{\mathbf{A}\}w) dx dy \tag{2.99}$$

and that the stiffness matrix becomes

$$[\mathbf{k}_m] = \int \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy \tag{2.100}$$

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

A typical value of the stiffness matrix is given by

$$k_{m i,j} = D \int \int \left\{ \frac{\partial^2 N_i}{\partial x^2} \frac{\partial^2 N_j}{\partial x^2} + \nu \frac{\partial^2 N_i}{\partial x^2} \frac{\partial^2 N_j}{\partial y^2} + \nu \frac{\partial^2 N_j}{\partial x^2} \frac{\partial^2 N_i}{\partial y^2} \right. \\ \left. + \frac{\partial^2 N_i}{\partial y^2} \frac{\partial^2 N_j}{\partial y^2} + 2(1 - \nu) \frac{\partial^2 N_i}{\partial x \partial y} \frac{\partial^2 N_j}{\partial x \partial y} \right\} dx dy \quad (2.101)$$

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

## 2.15 Summary of Element Equations for Solids

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

$$[\mathbf{k}_m] = \int \int [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy \quad (2.102)$$

and the element mass matrix

$$[\mathbf{m}_m] = \rho \int \int [\mathbf{N}]^T [\mathbf{N}] dx dy \quad (2.103)$$

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

$$1. \text{ Static equilibrium, } [\mathbf{k}_m]\{\mathbf{u}\} = \{\mathbf{f}\} \quad (2.104)$$

$$2. \text{ Eigenvalue, } [\mathbf{k}_m]\{\mathbf{a}\} - \omega^2 [\mathbf{m}_m]\{\mathbf{a}\} = \{\mathbf{0}\} \quad (2.105)$$

$$3. \text{ Propagation, } [\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{m}_m] \left\{ \frac{d^2 \mathbf{u}}{dt^2} \right\} = \{\mathbf{f}(t)\} \quad (2.106)$$

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

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

## 2.16 Flow of Fluids: Navier–Stokes Equations

We shall be concerned only with the equations governing the motion of viscous, incompressible fluids. These equations are widely developed elsewhere (e.g., Schlichting, 1960).

For two dimensions, preserving an analogy with previous sections on two-dimensional solids,  $u$  and  $v$  now become velocities in the  $x$ - and  $y$ -directions, respectively and  $\rho$  is the mass density. Also as before,  $F_x$  and  $F_y$  are body forces in the appropriate directions.

Conservation of mass leads to

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x}(\rho u) + \frac{\partial}{\partial y}(\rho v) = 0 \quad (2.107)$$

but due to incompressibility this may be reduced to

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \quad (2.108)$$

Conservation of momentum leads to

$$\begin{aligned} \rho \left( \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} \right) &= F_x + \left( \frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} \right) \\ \rho \left( \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} \right) &= F_y + \left( \frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} \right) \end{aligned} \quad (2.109)$$

where  $\sigma_x$ ,  $\sigma_y$  and  $\tau_{xy}$  are stress components as previously defined for solids.

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

$$\begin{aligned} \sigma_x &= -p - \frac{2\mu}{3} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + 2\mu \frac{\partial u}{\partial x} \\ \sigma_y &= -p - \frac{2\mu}{3} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + 2\mu \frac{\partial v}{\partial y} \\ \tau_{xy} &= \mu \left( \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \end{aligned} \quad (2.110)$$

where  $p$  is the fluid pressure.

Combining equations (2.109)–(2.110) in the absence of body forces, a form of the ‘Navier–Stokes’ equations can be written

$$\begin{aligned} \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} &= \frac{1}{\rho} F_x - \frac{1}{\rho} \frac{\partial p}{\partial x} + \frac{1}{3} \frac{\mu}{\rho} \frac{\partial}{\partial x} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + \frac{\mu}{\rho} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \\ \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} &= \frac{1}{\rho} F_y - \frac{1}{\rho} \frac{\partial p}{\partial y} + \frac{1}{3} \frac{\mu}{\rho} \frac{\partial}{\partial y} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + \frac{\mu}{\rho} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \end{aligned} \quad (2.111)$$

On introduction of the incompressibility condition (2.108), these can be further simplified to

$$\begin{aligned} \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} &= \frac{1}{\rho} F_x - \frac{1}{\rho} \frac{\partial p}{\partial x} + \frac{\mu}{\rho} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \\ \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} &= \frac{1}{\rho} F_y - \frac{1}{\rho} \frac{\partial p}{\partial y} + \frac{\mu}{\rho} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \end{aligned} \quad (2.112)$$

For steady-state conditions the terms  $\partial u / \partial t$  and  $\partial v / \partial t$  can be dropped, resulting in ‘coupled’ equations in the ‘primitive’ variables  $u$ ,  $v$  and  $p$ . The equations are also, in contrast to those of solid elasticity, non-linear due to the presence of products like  $u(\partial u / \partial x)$ .

Finally, the steady-state equations to be solved are

$$\begin{aligned} u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} + \frac{1}{\rho} \frac{\partial p}{\partial x} - \frac{\mu}{\rho} \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) &= 0 \\ u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} + \frac{1}{\rho} \frac{\partial p}{\partial y} - \frac{\mu}{\rho} \left( \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) &= 0 \end{aligned} \quad (2.113)$$

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

For the purposes of integration during the Galerkin procedure, the terms  $u$  and  $v$  in (2.113) are set equal to the constants  $u = \bar{u} = [\mathbf{N}]\{\mathbf{u}_0\}$  and  $v = \bar{v} = [\mathbf{N}]\{\mathbf{v}_0\}$ , where  $\{\mathbf{u}_0\}$  and  $\{\mathbf{v}_0\}$  are estimates of the nodal velocities (see Chapter 9).

After substitution of the trial solutions,

$$\begin{aligned} \bar{u} \frac{\partial}{\partial x} [\mathbf{N}]\{\mathbf{u}\} + \bar{v} \frac{\partial}{\partial y} [\mathbf{N}]\{\mathbf{u}\} + \frac{1}{\rho} \frac{\partial}{\partial x} [\mathbf{N}]\{\mathbf{p}\} - \frac{\mu}{\rho} \frac{\partial^2}{\partial x^2} [\mathbf{N}]\{\mathbf{u}\} - \frac{\mu}{\rho} \frac{\partial^2}{\partial y^2} [\mathbf{N}]\{\mathbf{u}\} &= \{\mathbf{0}\} \\ \bar{u} \frac{\partial}{\partial x} [\mathbf{N}]\{\mathbf{v}\} + \bar{v} \frac{\partial}{\partial y} [\mathbf{N}]\{\mathbf{v}\} + \frac{1}{\rho} \frac{\partial}{\partial y} [\mathbf{N}]\{\mathbf{p}\} - \frac{\mu}{\rho} \frac{\partial^2}{\partial x^2} [\mathbf{N}]\{\mathbf{v}\} - \frac{\mu}{\rho} \frac{\partial^2}{\partial y^2} [\mathbf{N}]\{\mathbf{v}\} &= \{\mathbf{0}\} \end{aligned} \quad (2.114)$$

Multiplying by the weighting functions and integrating as usual yields

$$\begin{aligned} \int \int [\mathbf{N}]^T \bar{u} \frac{\partial}{\partial x} [\mathbf{N}]\{\mathbf{u}\} dx \ dy + \int \int [\mathbf{N}]^T \bar{v} \frac{\partial}{\partial y} [\mathbf{N}]\{\mathbf{u}\} dx \ dy + \frac{1}{\rho} \int \int [\mathbf{N}]^T \frac{\partial}{\partial x} [\mathbf{N}]\{\mathbf{p}\} dx \ dy \\ - \frac{\mu}{\rho} \int \int [\mathbf{N}]^T \frac{\partial^2}{\partial x^2} [\mathbf{N}]\{\mathbf{u}\} dx \ dy - \frac{\mu}{\rho} \int \int [\mathbf{N}]^T \frac{\partial^2}{\partial y^2} [\mathbf{N}]\{\mathbf{u}\} dx \ dy &= \{\mathbf{0}\} \\ \int \int [\mathbf{N}]^T \bar{u} \frac{\partial}{\partial x} [\mathbf{N}]\{\mathbf{v}\} dx \ dy + \int \int [\mathbf{N}]^T \bar{v} \frac{\partial}{\partial y} [\mathbf{N}]\{\mathbf{v}\} dx \ dy + \frac{1}{\rho} \int \int [\mathbf{N}]^T \frac{\partial}{\partial y} [\mathbf{N}]\{\mathbf{p}\} dx \ dy \\ - \frac{\mu}{\rho} \int \int [\mathbf{N}]^T \frac{\partial^2}{\partial x^2} [\mathbf{N}]\{\mathbf{v}\} dx \ dy - \frac{\mu}{\rho} \int \int [\mathbf{N}]^T \frac{\partial^2}{\partial y^2} [\mathbf{N}]\{\mathbf{v}\} dx \ dy &= \{\mathbf{0}\} \end{aligned} \quad (2.115)$$

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

$$\begin{aligned} \int \int [\mathbf{N}]^T \bar{u} \frac{\partial}{\partial x} [\mathbf{N}] dx \ dy \{\mathbf{u}\} + \int \int [\mathbf{N}]^T \bar{v} \frac{\partial}{\partial y} [\mathbf{N}] dx \ dy \{\mathbf{u}\} + \frac{1}{\rho} \int \int [\mathbf{N}]^T \frac{\partial}{\partial x} [\mathbf{N}] dx \ dy \{\mathbf{p}\} \\ + \frac{\mu}{\rho} \int \int \frac{\partial}{\partial x} [\mathbf{N}]^T \frac{\partial}{\partial x} [\mathbf{N}] dx \ dy \{\mathbf{u}\} + \frac{\mu}{\rho} \int \int \frac{\partial}{\partial y} [\mathbf{N}]^T \frac{\partial}{\partial y} [\mathbf{N}] dx \ dy \{\mathbf{u}\} &= \{\mathbf{0}\} \end{aligned}$$

$$\begin{aligned} & \int \int [\mathbf{N}]^T \bar{u} \frac{\partial}{\partial x} [\mathbf{N}] dx \, dy \{ \mathbf{v} \} + \int \int [\mathbf{N}]^T \bar{v} \frac{\partial}{\partial y} [\mathbf{N}] dx \, dy \{ \mathbf{v} \} + \frac{1}{\rho} \int \int [\mathbf{N}]^T \frac{\partial}{\partial y} [\mathbf{N}] dx \, dy \{ \mathbf{p} \} \\ & + \frac{\mu}{\rho} \int \int \frac{\partial}{\partial x} [\mathbf{N}]^T \frac{\partial}{\partial x} [\mathbf{N}] dx \, dy \{ \mathbf{v} \} + \frac{\mu}{\rho} \int \int \frac{\partial}{\partial y} [\mathbf{N}]^T \frac{\partial}{\partial y} [\mathbf{N}] dx \, dy \{ \mathbf{v} \} = \{ \mathbf{0} \} \end{aligned} \quad (2.116)$$

The set of equations is completed by the continuity condition,

$$\int \int [\mathbf{N}]^T \left( \frac{\partial}{\partial x} [\mathbf{N}] \{ \mathbf{u} \} + \frac{\partial}{\partial y} [\mathbf{N}] \{ \mathbf{v} \} \right) dx \, dy = \{ \mathbf{0} \} \quad (2.117)$$

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

$$\begin{bmatrix} [\mathbf{c}_{11}] & [\mathbf{c}_{12}] & [\mathbf{c}_{13}] \\ [\mathbf{c}_{21}] & [\mathbf{c}_{22}] & [\mathbf{c}_{23}] \\ [\mathbf{c}_{31}] & [\mathbf{c}_{32}] & [\mathbf{c}_{33}] \end{bmatrix} \begin{Bmatrix} \mathbf{u} \\ \mathbf{p} \\ \mathbf{v} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \quad (2.118)$$

where

$$\begin{aligned} [\mathbf{c}_{11}] &= \int \int \left( [\mathbf{N}]^T \bar{u} \frac{\partial}{\partial x} [\mathbf{N}] + [\mathbf{N}]^T \bar{v} \frac{\partial}{\partial y} [\mathbf{N}] + \frac{\mu}{\rho} \frac{\partial}{\partial x} [\mathbf{N}]^T \frac{\partial}{\partial x} [\mathbf{N}] + \frac{\mu}{\rho} \frac{\partial}{\partial y} [\mathbf{N}]^T \frac{\partial}{\partial y} [\mathbf{N}] \right) dx \, dy \\ [\mathbf{c}_{12}] &= \frac{1}{\rho} \int \int [\mathbf{N}]^T \frac{\partial}{\partial x} [\mathbf{N}] dx \, dy \\ [\mathbf{c}_{13}] &= [\mathbf{0}] \\ [\mathbf{c}_{21}] &= \int \int [\mathbf{N}]^T \frac{\partial}{\partial x} [\mathbf{N}] dx \, dy \\ [\mathbf{c}_{22}] &= [\mathbf{0}] \\ [\mathbf{c}_{23}] &= \int \int [\mathbf{N}]^T \frac{\partial}{\partial y} [\mathbf{N}] dx \, dy \\ [\mathbf{c}_{31}] &= [\mathbf{0}] \\ [\mathbf{c}_{32}] &= \frac{1}{\rho} \int \int [\mathbf{N}]^T \frac{\partial}{\partial y} [\mathbf{N}] dx \, dy \\ [\mathbf{c}_{33}] &= [\mathbf{c}_{11}] \end{aligned} \quad (2.119)$$

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

## 2.17 Simplified Flow Equations

In many practical instances it may not be necessary to solve the complete coupled system described in the previous section. The pressure  $p$  can be eliminated from (2.112) and if vorticity is defined as

$$\omega = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} \quad (2.120)$$

this results in a single equation:

$$\frac{\partial \omega}{\partial t} + u \frac{\partial \omega}{\partial x} + v \frac{\partial \omega}{\partial y} = \frac{\mu}{\rho} \left( \frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2} \right) \quad (2.121)$$

Defining a stream function  $\psi$  such that

$$\begin{aligned} u &= \frac{\partial \psi}{\partial y} \\ v &= -\frac{\partial \psi}{\partial x} \end{aligned} \quad (2.122)$$

an alternative coupled system involving  $\psi$  and  $\omega$  can be devised, given here for steady-state conditions,

$$\begin{aligned} \frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} &= \omega \\ \frac{\mu}{\rho} \left( \frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2} \right) &= \frac{\partial \psi}{\partial y} \frac{\partial \omega}{\partial x} - \frac{\partial \psi}{\partial x} \frac{\partial \omega}{\partial y} \end{aligned} \quad (2.123)$$

This clearly has the advantage that only two unknowns are involved rather than the previous three. However, the solution of (2.123) is still a relatively complicated process and flow problems are sometimes solved via equation (2.121) alone, assuming that  $u$  and  $v$  can be approximated by some independent means or measured. In this form, equation (2.121) is an example of the ‘diffusion–convection’ equation, the second-order space derivatives corresponding to a ‘diffusion’ process and the first-order ones to a ‘convection’ process. The equation arises in various areas of engineering, for example sediment transport and pollutant disposal (Smith, 1976, 1979).

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

$$\frac{\partial \omega}{\partial t} = \frac{\mu}{\rho} \left( \frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2} \right) \quad (2.124)$$

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

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

$$\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2} = 0 \quad (2.125)$$

leaving the familiar ‘Laplace’ equation (or ‘Poisson’ equation if the right-hand side is non-zero). In the following sections, finite element formulations of these simplified flow equations are described, in order of increasing complexity.

### 2.17.1 Steady State

The form of Laplace’s equation (2.125) which arises in geomechanics, for example concerning 2D groundwater flow beneath a water-retaining structure or in an aquifer (Muskat, 1937) is

$$k_x \frac{\partial^2 \phi}{\partial x^2} + k_y \frac{\partial^2 \phi}{\partial y^2} = 0 \quad (2.126)$$

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

$$[\mathbf{k}_c]\{\phi\} = \{\mathbf{q}\} \quad (2.127)$$

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

With the usual finite element discretisation,

$$\tilde{\phi} = [\mathbf{N}]\{\phi\} \quad (2.128)$$

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

$$\int \int \left( k_x \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} + k_y \frac{\partial N_i}{\partial y} \frac{\partial N_j}{\partial y} \right) dx dy \quad (2.129)$$

A convenient way of expressing the matrix  $[\mathbf{k}_c]$  in (2.127) is

$$[\mathbf{k}_c] = \int \int [\mathbf{T}]^T [\mathbf{K}] [\mathbf{T}] dx dy \quad (2.130)$$

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

$$[\mathbf{K}] = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix} \quad (2.131)$$

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

$$[\mathbf{T}] = \begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial x} & \frac{\partial N_4}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \frac{\partial N_3}{\partial y} & \frac{\partial N_4}{\partial y} \end{bmatrix} \quad (2.132)$$

The similarity between (2.132) for a fluid and (2.69) for a solid enables the corresponding programs to look similar in spite of the governing differential equations being quite different. This unity of treatment is utilised in describing the programming techniques in Chapter 3.

Finally, it is worth noting that (2.130) can also be arrived at from energy considerations. The equivalent energy statement is that the integral

$$\int \int \left[ \frac{1}{2} k_x \left( \frac{\partial \phi}{\partial x} \right)^2 + \frac{1}{2} k_y \left( \frac{\partial \phi}{\partial y} \right)^2 \right] dx dy \quad (2.133)$$

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

Example solutions to steady-state problems described by (2.126) are given in Chapter 7. Three-dimensional problems are also solved in Chapter 12.

### 2.17.2 Transient State

Transient conditions must be analysed in many physical situations, for example in the case of Terzaghi ‘consolidation’ in soil mechanics or transient heat conduction. The governing diffusion equation for excess pore pressure  $u_w$  in 1D takes the form

$$c_z \frac{\partial^2 u_w}{\partial z^2} = \frac{\partial u_w}{\partial t} \quad (2.134)$$

in which  $c_z$  is the vertical coefficient of consolidation (or thermal diffusivity if dealing with transient heat flow).

In order to correctly model settlement and excess pore pressure dissipation in layered soils, it is necessary to break the coefficient of consolidation into its constituent parts (e.g., Huang and Griffiths, 2010), given as

$$c_z = \frac{k}{m_v \gamma_w} \quad (2.135)$$

where  $k$  and  $m_v$  are the soil permeability and coefficient of volume compressibility, and  $\gamma_w$  is the unit weight of water. In this case (2.134) is written as

$$\frac{k}{\gamma_w} \frac{\partial^2 u_w}{\partial z^2} = m_v \frac{\partial u_w}{\partial t} \quad (2.136)$$

In 2D, (2.134) takes the form

$$c_x \frac{\partial^2 u_w}{\partial x^2} + c_y \frac{\partial^2 u_w}{\partial y^2} = \frac{\partial u_w}{\partial t} \quad (2.137)$$

where  $c_x$  and  $c_y$  are the coefficients of consolidation in the  $x$ - and  $y$ -directions. Discretisation of the left-hand side of (2.137) or (2.134) clearly follows that of (2.126), while the time derivative will be associated with a matrix of the ‘mass matrix’ type from (2.75), without the multiple  $\rho$ . Hence the discretised system is

$$[\mathbf{k}_c]\{\mathbf{u}_w\} + [\mathbf{m}_m] \left\{ \frac{d\mathbf{u}_w}{dt} \right\} = \{\mathbf{0}\} \quad (2.138)$$

where  $\{\mathbf{u}_w\}$  are the nodal values of  $u_w$ . If the partitioning shown in (2.136) is used, the terms  $k/\gamma_w$  and  $m_v$  are absorbed in  $[\mathbf{k}_c]$  and  $[\mathbf{m}_m]$ , respectively.

The set of first-order, ordinary differential equations given by (2.138) can be solved by many methods, the simplest of which discretise the time derivative by finite differences. The algorithms are described in Chapter 3, with example solutions in Chapters 8 and 12.

### 2.17.3 Convection

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

$$c_x \frac{\partial^2 \phi}{\partial x^2} + c_y \frac{\partial^2 \phi}{\partial y^2} - u \frac{\partial \phi}{\partial x} - v \frac{\partial \phi}{\partial y} = \frac{\partial \phi}{\partial t} \quad (2.139)$$

where  $\phi$  can be interpreted as a ‘concentration’ and  $u$  and  $v$  are the fluid velocity components in the  $x$ - and  $y$ -directions [compare equation (2.121)].

The extra convection terms  $-u\partial\phi/\partial x$  and  $-v\partial\phi/\partial y$  compared with (2.137) lead, as shown in Table 2.1, to unsymmetric components of the resulting matrix of the type

$$\int \int \left( -u N_i \frac{\partial N_j}{\partial x} - v N_i \frac{\partial N_j}{\partial y} \right) dx dy \quad (2.140)$$

which must be added to the symmetric diffusion components given in (2.129). When this has been done, equilibrium equations like (2.127) or transient equations like (2.138) are regained.

Mathematically, equation (2.139) is a differential equation which is not self-adjoint (Berg, 1962), due to the presence of the first-order spatial derivatives. From a finite element point of view, equations which are not self-adjoint will always lead to unsymmetrical matrices.

A second consequence of non-self-adjoint equations is that there is no energy formulation equivalent to (2.133). It is clearly a benefit of the Galerkin approach that it can be used for all types of equation and is not restricted to self-adjoint systems.

Equation (2.139) can be rendered self-adjoint by using the transformation

$$h = \phi \exp \left( \frac{ux}{2c_x} \right) \exp \left( \frac{vy}{2c_y} \right) \quad (2.141)$$

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

Equation (2.139) and the use of (2.141) are described in Chapter 8.

## 2.18 Further Coupled Equations: Biot Consolidation

Thus far in this chapter, analyses of solids and fluids have been considered separately. However, Biot (1941) formulated the theory of coupled solid–fluid interaction which finds application in soil mechanics (Smith and Hobbs, 1976). The soil skeleton is treated as a porous elastic solid and the laminar pore fluid is coupled to the solid by the conditions of equilibrium and continuity.

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

$$\begin{aligned} \frac{\partial \sigma'_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial u_w}{\partial x} &= 0 \\ \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma'_y}{\partial y} + \frac{\partial u_w}{\partial y} &= 0 \end{aligned} \quad (2.142)$$

where  $\sigma'_x = \sigma_x - u_w$ , etc. are ‘effective’ stresses.

Assuming plane-strain conditions and small strains, and following the usual sequence of operations for a displacement method, the stress terms in equation (2.142) can be

eliminated in terms of displacements to give (e.g., Griffiths, 1994)

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

$$\frac{E'(1-\nu')}{(1+\nu')(1-2\nu')} \left[ \frac{1}{2(1-\nu')} \frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2 v}{\partial y^2} + \frac{(1-2\nu')}{2(1-\nu')} \frac{\partial^2 v}{\partial x^2} \right] + \frac{\partial u_w}{\partial y} = 0 \quad (2.143)$$

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

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

$$\frac{\partial}{\partial t} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + \frac{k_x}{\gamma_w} \frac{\partial^2 u_w}{\partial x^2} + \frac{k_y}{\gamma_w} \frac{\partial^2 u_w}{\partial y^2} = 0 \quad (2.144)$$

where  $k_x$  and  $k_y$  are the material permeabilities in the  $x$ - and  $y$ -directions, and  $\gamma_w$  is the unit weight of water.

Equations (2.143) and (2.144) represent the coupled ‘Biot’ equations for a 2D poroelastic material. A solution to these equations will enable the displacements  $u$ ,  $v$  and excess pore pressure  $u_w$  to be estimated at spatial location  $(x, y)$  at any time  $t$ .

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

$$\begin{aligned} \tilde{\mathbf{u}} &= [\mathbf{N}]\{\mathbf{u}\} \\ \tilde{\mathbf{v}} &= [\mathbf{N}]\{\mathbf{v}\} \\ \tilde{u}_w &= [\mathbf{N}]\{\mathbf{u}_w\} \end{aligned} \quad (2.145)$$

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

When discretisation and the Galerkin process are completed, (2.143) and (2.144) lead to the pair of equilibrium and continuity equations

$$\begin{aligned} [\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{c}]\{\mathbf{u}_w\} &= \{\mathbf{f}\} \\ [\mathbf{c}]^T \left\{ \frac{d\mathbf{u}}{dt} \right\} - [\mathbf{k}_c]\{\mathbf{u}_w\} &= \{\mathbf{0}\} \end{aligned} \quad (2.146)$$

where, for a 4-node element,

$$\{\mathbf{u}\} = \begin{pmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{pmatrix} \text{ and } \{\mathbf{u}_w\} = \begin{pmatrix} u_{w1} \\ u_{w2} \\ u_{w3} \\ u_{w4} \end{pmatrix} \quad (2.147)$$

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

$$\int \int \frac{\partial N_i}{\partial x} N_j dx dy \quad (2.148)$$

and  $\{\mathbf{f}\}$  is the external loading vector. After assembly into global matrices, equations (2.146) must be integrated in time by some method such as finite differences and this is described further in Chapter 3. Examples of such solutions in practice are given in Chapter 9.

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

## 2.19 Conclusions

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

$$[\mathbf{k}_m]\{\mathbf{u}\} = \{\mathbf{f}\} \quad (2.149)$$

or

$$[\mathbf{k}_c]\{\phi\} = \{\mathbf{q}\} \quad (2.150)$$

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

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

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

## References

- Bathe KJ and Wilson EL 1996 *Numerical Methods in Finite Element Analysis*, 3rd edn. Prentice-Hall, Englewood Cliffs, NJ.
- Berg PN 1962 *Calculus of Variations*. McGraw-Hill, London.
- Biot MA 1941 General theory of three-dimensional consolidation. *J Appl Phys* **12**, 155–164.
- Cook RD, Malkus DS, Plesha ME and Witt RJ 2002 *Concepts and Applications of Finite Element Analysis*, 4th edn. John Wiley & Sons, Chichester.
- Finlayson BA 1972 *The Method of Weighted Residuals and Variational Principles*. Academic Press, New York.
- Griffiths DV 1989 Advantages of consistent over lumped methods for analysis of beams on elastic foundations. *Comm Appl Numer Methods* **5**(1), 53–60.
- Griffiths DV 1994 Coupled analyses in geomechanics In *Visco-plastic Behavior of Geomaterials* (eds. Cristescu ND and Gioda G). Springer-Verlag Wien, New York, pp. 245–317.

- Griffiths DV and Smith IM 2006 *Numerical Methods for Engineers*, 2nd edn. Chapman & Hall/CRC Press, Boca Raton, FL.
- Horne MR and Merchant W 1965 *The Stability of Frames*. Pergamon Press, Oxford.
- Huang J and Griffiths DV 2010 One-dimensional consolidation theories for layered soil and coupled and uncoupled solutions by the finite-element method. *Géotechnique* **60**(9), 709–713.
- Jennings A and McKeown JJ 1992 *Matrix Computation*, 2nd edn. John Wiley & Sons, Chichester.
- Leckie FA and Lindberg GM 1963 The effect of lumped parameters on beam frequencies. *Aeronaut Q* **14**, 234.
- Livesley RK 1975 *Matrix Methods of Structural Analysis*. Pergamon Press, Oxford.
- Muskat M 1937 *The Flow of Homogeneous Fluids through Porous Media*. McGraw-Hill, London.
- Rao SS 2010 *The Finite Element Method in Engineering*, 5th edn. Butterworth-Heinemann, Woburn, MA.
- Schllichting H 1960 *Boundary Layer Theory*. McGraw-Hill, London.
- Smith IM 1976 Integration in time of diffusion and diffusion–convection equations. *Finite Elements in Water Resources*, Vol. 1. Pentech Press, Plymouth, pp. 3–20.
- Smith IM 1979a The diffusion–convection equation. In *Summary of Numerical Methods for Partial Differential Equations*. Oxford University Press, Oxford, pp. 195–211.
- Smith IM and Hobbs R 1976 Biot analysis of consolidation beneath embankments. *Géotechnique* **26**, 149–171.
- Smith IM, Farraday RV and O'Connor BA 1973 Rayleigh–Ritz and Galerkin finite elements for diffusion–convection problems. *Water Resour Res* **9**(3), 593–606.
- Strang G and Fix GJ 2008 *An Analysis of the Finite Element Method*. Wellesley-Cambridge, Wellesley, MA.
- Szabo BA and Lee GC 1969 Derivation of stiffness matrices for problems in plane elasticity by the Galerkin method. *Int J Numer Methods Eng* **1**, 301.
- Taig IC 1961 Structural analysis by the matrix displacement method. Technical Report SO17, English Electric Aviation Report, Preston.
- Taylor C and Hughes TG 1981 *Finite Elements Programming of the Navier–Stokes Equation*. Pineridge Press, Swansea.
- Timoshenko SP and Goodier JN 1982 *Theory of Elasticity*, 3rd edn. McGraw-Hill International Editions, Singapore.
- Timoshenko SP and Woinowsky-Krieger S 1959 *Theory of Plates and Shells*. McGraw-Hill, New York.
- Zienkiewicz OC, Taylor RL and Zhu JZ 2005 *The Finite Element Method*, Vol. 1, 6th edn. McGraw-Hill, London.



# 3

# Programming Finite Element Computations

## 3.1 Introduction

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

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

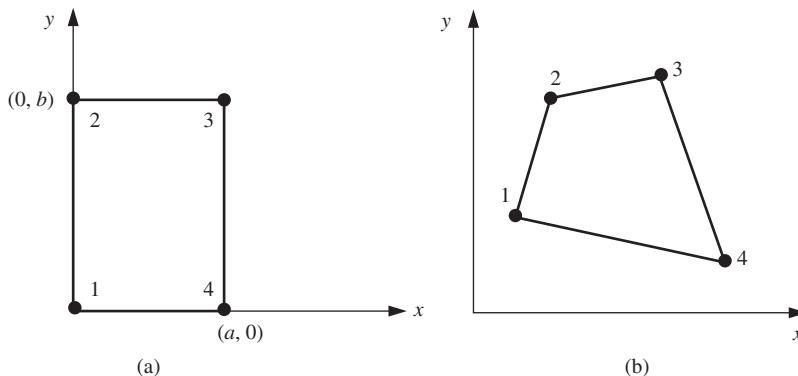
Second, we have so far considered only a single element, whereas useful solutions will normally be obtained by many elements, usually from hundreds to millions in practice, joined together at the nodes. Also, various types of boundary conditions may be prescribed which constrain the solution in some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

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

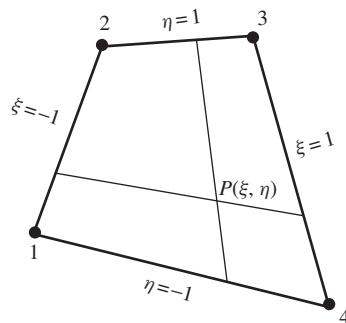
## 3.2 Local Coordinates for Quadrilateral Elements

Figure 3.1 shows two types of plane 4-node quadrilateral elements. The shape functions for the rectangle (Figure 3.1(a)) were shown to be given by equation (2.60), namely  $N_1 = (1 - x/a)(1 - y/b)$  and so on. If it is attempted to construct similar shape functions in the ‘global’ coordinates  $(x, y)$  for the general quadrilateral (Figure 3.1(b)), very complex algebraic expressions will result, which are best generated by computer algebra packages (e.g., Griffiths, 1994b, 2004).

Traditionally the approach has been to work in a local coordinate system as shown in Figure 3.2, originally proposed by Taig (1961), and to evaluate resulting



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



**Figure 3.2** Local coordinate system for quadrilateral elements

integrals numerically. The general point  $P(\xi, \eta)$  within the quadrilateral is located at the intersection of two lines which cut opposite sides of the quadrilateral in equal proportions. For reasons associated with subsequent numerical integrations it proves to be convenient to ‘normalise’ the coordinates so that side 12 has  $\xi = -1$ , side 34 has  $\xi = 1$ , side 41 has  $\eta = -1$  and side 23 has  $\eta = 1$ . In this system the intersection of the bisectors of opposite sides of the quadrilateral is the point  $(0, 0)$ , while the corners 1, 2, 3 and 4 are  $(-1, -1)$ ,  $(-1, 1)$ ,  $(1, 1)$  and  $(1, -1)$ , respectively.

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

$$\begin{aligned}
 N_1 &= \frac{1}{4}(1 - \xi)(1 - \eta) \\
 N_2 &= \frac{1}{4}(1 - \xi)(1 + \eta) \\
 N_3 &= \frac{1}{4}(1 + \xi)(1 + \eta) \\
 N_4 &= \frac{1}{4}(1 + \xi)(1 - \eta)
 \end{aligned} \tag{3.1}$$

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

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

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

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

In the previous chapter [e.g., equations (2.80) and (2.132)] it was shown that element properties involve not only  $[\mathbf{N}]$  but also their derivatives with respect to the global coordinates ( $x, y$ ) which appear in matrices such as  $[\mathbf{B}]$  and  $[\mathbf{T}]$ . Further, products of these quantities need to be integrated over the element area or volume.

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

$$\left\{ \begin{array}{l} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{array} \right\} = \left[ \begin{array}{cc} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{array} \right] \left\{ \begin{array}{l} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{array} \right\} = [\mathbf{J}] \left\{ \begin{array}{l} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{array} \right\} \quad (3.3)$$

or

$$\left\{ \begin{array}{l} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{array} \right\} = [\mathbf{J}]^{-1} \left\{ \begin{array}{l} \frac{\partial}{\partial \xi} \\ \frac{\partial}{\partial \eta} \end{array} \right\} \quad (3.4)$$

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

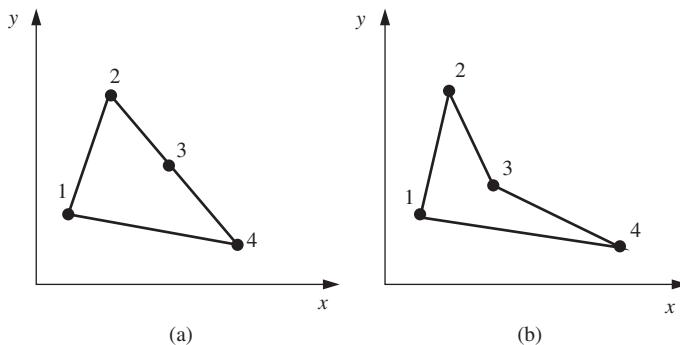
$$\iint dx dy = \int_{-1}^1 \int_{-1}^1 \det[\mathbf{J}] d\xi d\eta \quad (3.5)$$

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

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

### 3.2.1 Numerical Integration for Quadrilaterals

Although some integrals of this type can be evaluated analytically, this has traditionally been impractical for complicated functions, particularly in the general case when  $(\xi, \eta)$  become curvilinear (e.g., Ergatoudis *et al.*, 1968). In most finite element programs (3.5)



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

is evaluated numerically, using Gauss–Legendre quadrature over quadrilateral regions (Irons, 1966). The quadrature rules in two dimensions are all of the form

$$\int_{-1}^1 \int_{-1}^1 f(\xi, \eta) d\xi d\eta \approx \sum_{i=1}^n \sum_{j=1}^n w_i w_j f(\xi_i, \eta_j)$$

$$\approx \sum_{i=1}^{\text{nip}} W_i f(\xi, \eta)_i \quad (3.6)$$

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

The approximate equality in (3.6) is exact for cubic functions when  $n = 2$  and for quintics when  $n = 3$ . Usually one attempts to perform integrations over finite elements

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

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

exactly, but in special circumstances (Zienkiewicz *et al.*, 1971) ‘reduced’ integration, whereby integrals are evaluated approximately by decreasing  $n$  can improve the quality of solutions.

### 3.2.2 Analytical Integration for Quadrilaterals

Computer algebra systems (CASs) such as ‘REDUCE’ and ‘Maple’ enable algebraic expressions (e.g., the finite element shape functions) to be manipulated essentially ‘analytically’. Expressions can be differentiated, integrated, factorised and so on, leading to explicit formulations of element matrices avoiding the need for conventional numerical integration. Particularly for 3D elements, this approach can lead to substantial savings in integration times. A further point is that for some elements (e.g., a 14-node hexahedron described later in this chapter) the shape functions are so complex algebraically that it is doubtful if they could be isolated at all without the help of computer algebra.

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

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy \quad (3.7)$$

where  $[\mathbf{B}]$  and  $[\mathbf{D}]$  represent the strain–displacement and stress–strain matrices, respectively.

In the case of quadrilateral elements, if the element is rectangular with its sides parallel to the  $x$ - and  $y$ -axes, the term under the integral consists of simple polynomial terms which can be easily integrated in closed form by separation of the variables, resulting in compact terms like (2.63). In general, however, quadrilateral elements will lead to very complicated expressions under the integral sign which can only be tackled numerically (e.g., Griffiths 2004).

Noting that ‘2-point’ Gaussian quadrature, that is  $n_{ip} = 4$ , leads in most cases to accurate estimates of the stiffness matrix of a 4-node general quadrilateral, a compromise approach is to evaluate the contribution to the stiffness matrix coming from each of the four ‘Gauss-points’ algebraically and add them together, thus

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

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

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

The simplified algebraic expressions that form the stiffness matrix of the 4-node quadrilateral element by this method have been isolated, and form the basis of subroutine `stiff4` used in Program 11.6 of this book. A detailed description of the method is given in Griffiths (1994b).

The same technique can be applied to other element types (Cardoso, 1994) and other element matrices (e.g., 8-node quadrilaterals, 3D elements, mass, conductivity, etc.). For example, the technique is to be found again in Program 7.3, where the conductivity

matrix of a general 4-node quadrilateral element is computed algebraically using subroutine `seep4`.

A similar approach was used to create subroutine `b8e8` used in Programs 6.4, 6.10 and 6.11, which generates an algebraic version of the  $[\mathbf{B}]$  matrix for a general 8-node quadrilateral element, corresponding to any given local coordinate  $(\xi, \eta)$ .

### 3.3 Local Coordinates for Triangular Elements

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

$$L_3 = 1 - L_1 - L_2 \quad (3.9)$$

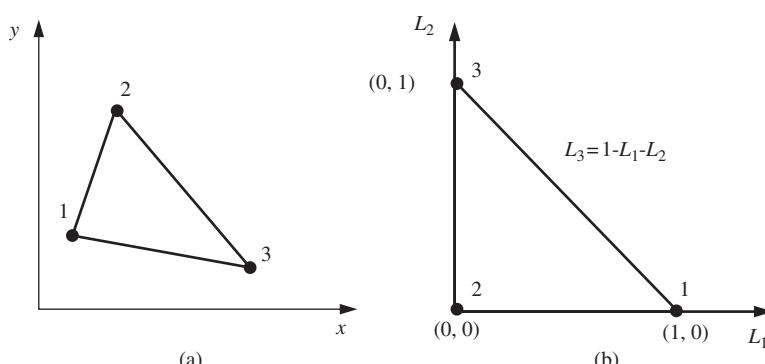
can sometimes be included to simplify the algebra.

For example, the shape functions for a 3-node (‘constant strain’) triangular element [Figure 3.4(b)] take the form

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

and as before the isoparametric property gives

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



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

Equations (3.3) and (3.4) in Section 3.2 still apply regarding the Jacobian matrix, but equation (3.5) must be modified for triangles to give

$$\iint dx \, dy = \int_0^1 \int_0^{1-L_1} \det|\mathbf{J}| \, dL_2 \, dL_1 \quad (3.12)$$

### 3.3.1 Numerical Integration for Triangles

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

$$\int_0^{L_1} \int_0^{1-L_1} f(L_1, L_2) \, dL_2 \, dL_1 \approx \sum_{i=1}^{\text{nip}} W_i f(L_1, L_2)_i \quad (3.13)$$

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

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

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

### 3.3.2 Analytical Integration for Triangles

Unlike quadrilateral elements, the stiffness matrix of triangular elements with straight sides can always be obtained exactly using analytical integration. While giving the same answer (within machine accuracy), the analytical formulations can lead to significant runtime efficiencies over the more conventional approaches which use numerical integration, as discussed in the previous section. Griffiths *et al.* (2009) noted improvements of up to a factor of 27 for 15-node triangles. Further savings were also made possible by the observation that some terms of the stiffness matrix of triangles are always equal to one another, while others equal zero regardless of the nodal coordinates.

**Table 3.2** Coordinates and weights in triangular integration formulae

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

Subroutines for generating the exact stiffness matrix of 3-, 6-, 10- and 15-node triangles, called `stiff3`, `stiff6`, `stiff10` and `stiff15`, respectively are included in the main library and described in Appendix D, however they are not used in any of the programs in this book in the interests of compactness.

### 3.4 Multi-Element Assemblies

Properties of elements in isolation have been shown to be given by matrix equations, for example the conductivity equation (2.127),

$$[\mathbf{k}_c]\{\phi\} = \{\mathbf{q}\} \quad (3.14)$$

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

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

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

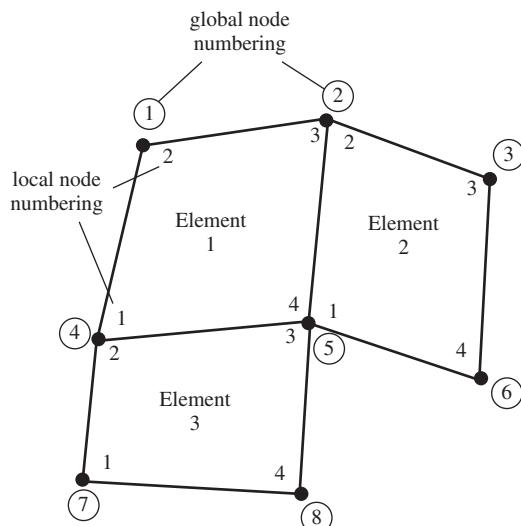


Figure 3.5 Mesh of quadrilateral elements

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

$k_{2,2}^1$	$k_{2,3}^1$	0	$k_{2,1}^1$	$k_{2,4}^1$	0	0	0
$k_{3,2}^1$	$k_{3,3}^1 + k_{2,2}^2$	$k_{2,3}^2$	$k_{3,1}^1$	$k_{3,4}^1 + k_{2,1}^2$	$k_{2,4}^2$	0	0
0	$k_{3,2}^2$	$k_{3,3}^2$	0	$k_{3,1}^2$	$k_{3,4}^2$	0	0
$k_{1,2}^1$	$k_{1,3}^1$	0	$k_{1,1}^1 + k_{2,2}^3$	$k_{1,4}^1 + k_{2,3}^3$	0	$k_{2,1}^3$	$k_{2,4}^3$
$k_{4,2}^1$	$k_{4,3}^1 + k_{1,2}^2$	$k_{1,3}^2$	$k_{4,1}^1 + k_{3,2}^3$	$k_{4,4}^1 + k_{1,1}^2 + k_{3,3}^3$	$k_{1,4}^2$	$k_{3,1}^3$	$k_{3,4}^3$
0	$k_{4,2}^2$	$k_{4,3}^2$	0	$k_{4,1}^2$	$k_{4,4}^2$	0	0
0	0	0	$k_{1,2}^3$	$k_{1,3}^3$	0	$k_{1,1}^3$	$k_{1,4}^3$
0	0	0	$k_{4,2}^3$	$k_{4,3}^3$	0	$k_{4,1}^3$	$k_{4,4}^3$

global matrix for Figure 3.5 is given in Table 3.3, where the superscripts refer to element numbers.

The global matrix equation can be written as

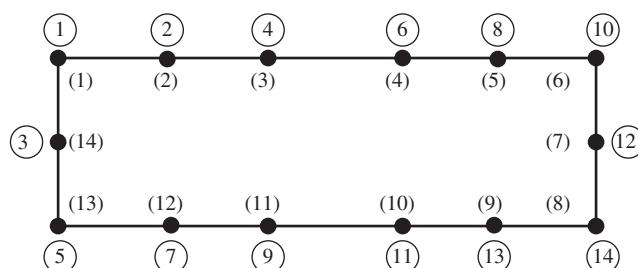
$$[\mathbf{K}_c]\{\Phi\} = \{\mathbf{Q}\} \quad (3.16)$$

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

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

The importance of efficient mesh numbering is illustrated for a mesh of line elements in Figure 3.6, where the scheme in parentheses has  $n_{band} = 13$  compared with the scheme using circles with  $n_{band} = 2$ .

If system symmetry exists it should also be taken into account. Using a constant bandwidth storage strategy, the system in Table 3.3 would require 40 storage locations (eight equations times five terms on each line). Greater efficiency can be achieved through ‘skyline’ storage (e.g., Bathe and Wilson, 1996), where the variability of the bandwidth is taken into account, requiring 28 storage locations in this case. Most of the programs



**Figure 3.6** Alternative mesh numbering schemes

described in this book make use of this variable bandwidth or ‘skyline’ storage strategy (see Figure 3.20 for examples of different storage strategies).

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

### 3.5 ‘Element-by-Element’ Techniques

Our purpose is to solve classes of problem, for example as summarised for solids by equations (2.104)–(2.106) for a single element by

$$1. \text{ Static equilibrium problems, } [\mathbf{k}_m] \{\mathbf{u}\} = \{\mathbf{f}\} \quad (3.17)$$

$$2. \text{ Eigenvalue problems, } [\mathbf{k}_m] \{\mathbf{a}\} - \omega^2 [\mathbf{m}_m] \{\mathbf{a}\} = \{\mathbf{0}\} \quad (3.18)$$

$$3. \text{ Propagation problems, } [\mathbf{k}_m] \{\mathbf{u}\} + [\mathbf{m}_m] \left\{ \frac{d^2 \mathbf{u}}{dt^2} \right\} = \{\mathbf{f}(t)\} \quad (3.19)$$

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

$$[\mathbf{K}_m] \{\mathbf{U}\} = \{\mathbf{F}\} \quad (3.20)$$

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

As problem sizes grow, this storage requirement becomes a burden, even on a modern computer. For meshes of 3D elements 100,000 equations are likely to have a semi-bandwidth of the order of 1000. Thus  $10^8$  ‘words’ of storage, typically 800 Mb, would be required to hold the coefficient matrix  $[\mathbf{K}_m]$ . If this space is not available, out-of-core techniques or ‘paging’ (see Section 1.3) cause a serious deterioration in analysis speeds.

For this reason, alternative solution strategies to Gaussian elimination have been sought, and there has been a resurgence of interest in iterative techniques for the solution of large systems like (3.20). Griffiths and Smith (2006) describe a number of algorithms of this type, the most popular for symmetric positive definite systems being based on the method of ‘conjugate gradients’ (Jennings and McKeown, 1992).

#### 3.5.1 Conjugate Gradient Method for Linear Equation Systems

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

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

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

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

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

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

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

$$\{\mathbf{Q}\} = \sum_{i=1}^{n_{els}} [\mathbf{k}_m]_i \{\mathbf{p}\}_i \quad (3.23)$$

where  $n_{els}$  is the number of elements,  $[\mathbf{k}_m]_i$  is the element stiffness matrix of the  $i$ th element and  $\{\mathbf{p}\}_i$  the appropriate part of  $\{\mathbf{P}\}$ , gathered as  $[p_7 \ p_4 \ p_5 \ p_8]^T$  for  $i = 3$  in Figure 3.5 and so on.

The storage required by such an algorithm, compared with the 800 Mb discussed earlier for a 3D system of 100,000 unknowns, would be an order of magnitude less, and would grow linearly with the increase in number of elements or equations rather than as the square. In practice, ‘preconditioning’ (e.g., Griffiths and Smith, 2006) can be used to accelerate convergence of some iterative processes for solving the ‘element-by-element’ version of (3.20). Iterative strategies for the solution of equations of the type given by (3.18) and (3.19) will also be described in due course.

### 3.5.2 Preconditioning

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

$$[\mathbf{P}] [\mathbf{K}_m] \{\mathbf{U}\} = [\mathbf{P}] \{\mathbf{F}\} \quad (3.24)$$

or

$$[\mathbf{K}_m][\mathbf{P}]\{\mathbf{U}\} = [\mathbf{P}]\{\mathbf{F}\} \quad (3.25)$$

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

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

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

### 3.5.3 Unsymmetric Systems

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

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

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

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

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

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

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

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

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

$$(a) \quad \begin{aligned} \{\mathbf{Q}\}_{k-1} &= [\mathbf{K}_m]\{\mathbf{P}\}_{k-1} \\ \{\mathbf{U}\}_{k-\frac{1}{2}} &= \{\mathbf{U}\}_{k-1} + \alpha_{k-1}\{\mathbf{P}\}_{k-1} \\ \text{where } \alpha_{k-1} &= \frac{\{\mathbf{R}\}_{k-1}^T\{\hat{\mathbf{R}}_0\}}{\{\mathbf{Q}\}_{k-1}^T\{\hat{\mathbf{R}}_0\}} \end{aligned} \quad (3.28)$$

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

$$(b) \quad \{\mathbf{S}\}_{k-\frac{1}{2}} = [\mathbf{K}_m]\{\mathbf{R}\}_{k-\frac{1}{2}}$$

$$\{\mathbf{U}\}_k = \{\mathbf{U}\}_{k-\frac{1}{2}} + \omega_k \{\mathbf{R}\}_{k-\frac{1}{2}}$$

$$\text{where} \quad \omega_k = \frac{\{\mathbf{R}\}_{k-\frac{1}{2}}^T \{\mathbf{S}\}_{k-\frac{1}{2}}}{\{\mathbf{S}\}_{k-\frac{1}{2}}^T \{\mathbf{S}\}_{k-\frac{1}{2}}} \quad (3.29)$$

$$\{\mathbf{R}\}_k = \{\mathbf{R}\}_{k-\frac{1}{2}} - \omega_k \{\mathbf{S}\}_{k-\frac{1}{2}}$$

$$\beta_k = \frac{\alpha_{k-1} \{\mathbf{R}\}_k^T \{\hat{\mathbf{R}}_0\}}{\omega_k \{\mathbf{R}\}_{k-1}^T \{\hat{\mathbf{R}}_0\}}$$

$$\{\mathbf{P}\}_k = \{\mathbf{R}\}_k + \beta_k \{\mathbf{P}\}_{k-1} - \beta_k \omega_k \{\mathbf{Q}\}_{k-1}$$

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

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

### 3.5.4 Symmetric Non-Positive Definite Equations

When Biot's equations for coupled consolidation [(2.143), (2.144)] are discretised by finite elements as (3.118) or (3.121), these systems will be seen to be symmetric but non-positive definite. Although a candidate solution algorithm is the minimum residual method (MINRES), Smith (2000) found that the diagonally preconditioned conjugate gradient method worked quite effectively and is used herein. However, the whole question of preconditioning is a developing area (e.g., Chan *et al.*, 2001).

### 3.5.5 Eigenvalue Systems

Again in an ‘element-by-element’ context we seek algorithms which have at their heart matrix–vector operations like (3.23) and vector or inner product operations which can readily be parallelised. Arnoldi methods (Arnoldi, 1951) fall into this category and have been incorporated in the freely available software package ARPACK (Lehoucq *et al.*, 1998), which will be used in Chapter 10. For symmetric systems the related Lanczos method (e.g., Bai *et al.*, 2000) can also be used, which is deceptively simple (Griffiths and Smith, 2006).

Suppose the eigenproblem (3.18) has been reduced to finding eigenvalues and eigenvectors of a symmetric positive definite matrix  $[\mathbf{A}]$  (the ‘hermitian eigenvalue problem’, Bai *et al.*, 2000). In the Lanczos method,  $[\mathbf{A}]$  is tridiagonalised by the following algorithm.

Choose a start vector  $\{\mathbf{Y}_1\}$ , and set  $\{\mathbf{Y}_0\}$  to  $\{\mathbf{0}\}$  and  $\beta_0$  to 0, where  $\{\alpha\}$  and  $\{\beta\}$  are the leading diagonal and off-diagonal, respectively of the tridiagonalisation.

Then for  $j=1, n_{eq}$  (the number of equations),

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

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

### 3.6 Incorporation of Boundary Conditions

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

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

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

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

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

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

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

$$\frac{\partial \phi}{\partial n} = 0 \quad (3.32)$$

$$\frac{\partial \phi}{\partial n} = C_1 \phi \quad (3.33)$$

$$\frac{\partial \phi}{\partial n} = C_2 \quad (3.34)$$

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

To be specific, consider a solution of the diffusion–convection equation (2.139) subject to boundary conditions (3.32), (3.33) and (3.34), respectively. When the second-order terms  $c_x \partial^2 \phi / \partial x^2$  and  $c_y \partial^2 \phi / \partial y^2$  are integrated by parts, boundary integrals of the type

$$\oint_S c_n [\mathbf{N}]^T \frac{\partial \phi}{\partial n} l_n dS \quad (3.35)$$

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

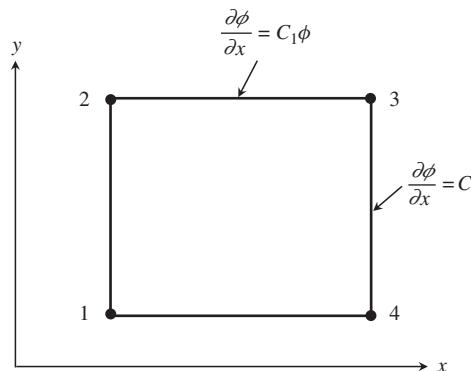
However, (3.33) gives rise to an extra integral, which for the boundary element shown in Figure 3.7 is

$$\int_j^k c_y [\mathbf{N}]^T C_1 \tilde{\phi} l_y dS \quad (3.36)$$

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

$$\frac{-C_1 c_y (x_3 - x_2)}{6} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (3.37)$$

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



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

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

$$\int_k^l c_x [\mathbf{N}]^T C_2 l_x dS \quad (3.38)$$

which is just a vector

$$\frac{C_2 c_x (y_3 - y_4)}{2} \begin{Bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{Bmatrix} \quad (3.39)$$

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

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

### 3.6.1 Convection Boundary Conditions

Analysis of temperature changes in conductive media regularly involve convection boundary conditions, where

$$\frac{\partial T}{\partial n} = h(T - T_\infty) \quad (3.40)$$

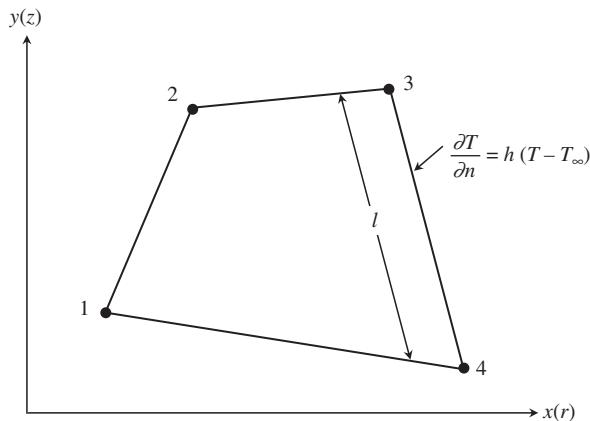
in which  $h$  is a material property called the convection heat transfer coefficient,  $T$  is the boundary temperature and  $T_\infty$  is the ambient temperature outside the body. Expansion of (3.40) shows that the convection boundary condition consists of both constant and linear terms from (3.33) and (3.34) of the form

$$\frac{\partial T}{\partial n} = C_1 T + C_2 \quad (3.41)$$

where  $C_1 = h$  and  $C_2 = -h T_\infty$ . Thus to achieve this boundary condition, changes to both the left- and right-hand sides of the element equations of the type shown in (3.37) and (3.39) will be necessary prior to assembly.

Consider the 4-node element shown in Figure 3.8, with a convection boundary condition imposed on the side of length  $l$  between nodes 3 and 4. For planar problems  $(x, y)$ , the left-hand-side matrix for assembly would be of the form

$$\frac{hl}{6} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix} \quad (3.42)$$



**Figure 3.8** Convection boundary conditions

and the right-hand-side vector would be

$$\frac{hT_{\infty}l}{2} \begin{Bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{Bmatrix} \quad (3.43)$$

For axisymmetric problems ( $r, z$ ), the corresponding matrix and vector would be

$$\frac{hl}{12} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3r_3 + r_4 & r_3 + r_4 \\ 0 & 0 & r_3 + r_4 & 3r_4 + r_3 \end{bmatrix} \quad (3.44)$$

and

$$\frac{hT_{\infty}l}{6} \begin{Bmatrix} 0 \\ 0 \\ 2r_3 + r_4 \\ r_3 + 2r_4 \end{Bmatrix} \quad (3.45)$$

Implementation of this boundary condition is described later in the text in Program 8.11.

### 3.7 Programming using Building Blocks

The programs in subsequent chapters are constituted from over 100 ‘building blocks’ in the form of FORTRAN functions and subroutines which perform the tasks of computing and integrating the element matrices, assembling these into system matrices if necessary and carrying out the appropriate equilibrium, eigenvalue or propagation calculations. In Chapter 12, additional functions and subroutines use MPI to handle the necessary communication between processors.

It is anticipated that users will elect to pre-compile all of the building blocks and to hold these permanently in a library. The library should then be automatically accessible

to the calling programs by means of a simple USE statement at the beginning of the program (see Chapter 1, Section 1.9.9).

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

### 3.7.1 Black Box Routines

Chapter 1, Section 1.9 summarised such features as whole-array operations and intrinsic array procedures, which mean that most simple array manipulations can be done using the power of the language itself and do not need to be user-supplied.

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

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

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

Several subroutines are associated with eigenvalue and eigenvector determination, for example for symmetric banded matrices, bandred tridiagonalises the matrix and bisect extracts all of the eigenvalues. It should be noted that these routines, although robust and accurate, can be inefficient both in storage requirements and in run-time and should not be used for solving very large problems, for which in any case it is unlikely that the full range of eigenmodes would be required. The various vector iteration methods (Bathe and Wilson, 1996) should be resorted to in such cases.

**Table 3.4** Subroutines for solution of linear algebraic equations

Method	Gauss	Cholesky	Gauss	Gauss
Storage	Symmetric half-band	Symmetric skyline	Symmetric skyline	Unsymmetric full band
Factorisation	bandred	sparin	sparin_gauss	gauss_band
Substitution	bacsbus	spabac	spabac_gauss	solve_band

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

When the Lanczos procedure is described in more detail, it will be found that, in common with its close relation the conjugate gradient procedure (Section 3.5.1), the method requires a matrix–vector product where the matrix is essentially the global system stiffness matrix, followed by a series of whole-vector operations. To save storage, the matrix–vector product can be done ‘element-by-element’ (3.23) and this feature is taken advantage of in Chapters 10 and 12.

Although simple matrix-by-vector multiplications can be accomplished by intrinsic procedure `MATMUL`, advantage is usually taken of the structure of global system matrix coefficients whenever possible. To allow for this, three special matrix-by-vector multiplication subroutines are provided as shown in Table 3.5, and further information on when these routines should be used is given in Section 3.8.

It is expected that users may pre- and post-process their data using independent programs.

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

### 3.7.2 Special Purpose Routines

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

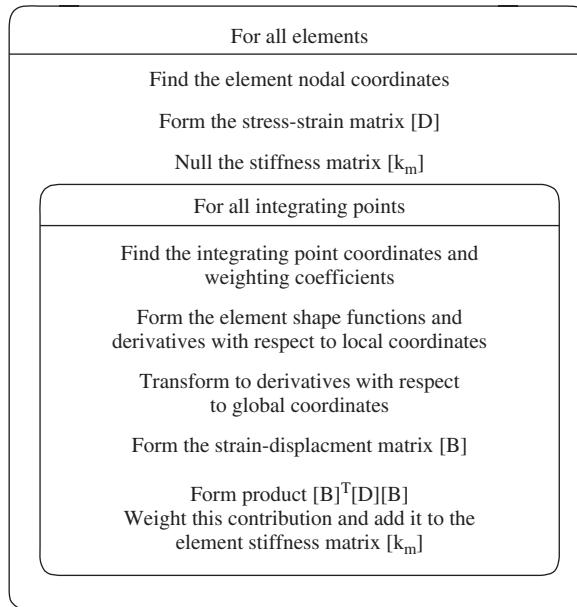
### 3.7.3 Plane Elastic Analysis using Quadrilateral Elements

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

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy \quad (3.46)$$

**Table 3.5** Subroutines for matrix–vector multiplication

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



**Figure 3.9** Structure chart for element stiffness matrix calculation assuming numerical integration

This formulation in the programs is described by the inner loop of the structure chart in Figure 3.9.

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

$$\text{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \quad (3.47)$$

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

$$\text{fun} = \left\{ \begin{array}{l} \frac{1}{4}(1-\xi)(1-\eta) \\ \frac{1}{4}(1-\xi)(1+\eta) \\ \frac{1}{4}(1+\xi)(1+\eta) \\ \frac{1}{4}(1+\xi)(1-\eta) \end{array} \right\} \quad (3.48)$$

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

$$\text{der} = \left[ \begin{array}{c} \frac{\partial \text{fun}^T}{\partial \xi} \\ \frac{\partial \text{fun}^T}{\partial \eta} \end{array} \right] \quad (3.49)$$

or

$$\text{der} = \frac{1}{4} \begin{bmatrix} -(1-\eta) & -(1+\eta) & (1+\eta) & (1-\eta) \\ -(1-\xi) & (1-\xi) & (1+\xi) & -(1+\xi) \end{bmatrix} \quad (3.50)$$

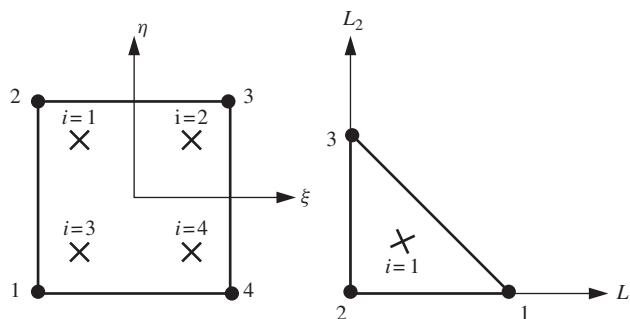
The information in (3.48) and (3.50) for a 4-node quadrilateral (`nod=4`) is formed by the subroutines `shape_fun` and `shape_der` for the specific Gaussian integration points  $(\xi, \eta)_i$  held in the array `points` where  $i$  runs from 1 to `nip`, the total number of sampling points specified in each element. Figure 3.10(a) shows the node numbering and the order in which the sampling (Gauss) points are scanned in a ‘two-point’ scheme. Since there are two integrating points in each coordinate direction, `nip=4` in this example. In all cases, points and their corresponding weights (Table 3.1) are found by the subroutine `sample`, where `nip` can take the values 1, 4 or 9 for quadrilaterals.

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

$$\begin{aligned} x &= \frac{1}{4}(1-\xi)(1-\eta)x_1 + \frac{1}{4}(1-\xi)(1+\eta)x_2 \\ &\quad + \frac{1}{4}(1+\xi)(1+\eta)x_3 + \frac{1}{4}(1+\xi)(1-\eta)x_4 \\ y &= \frac{1}{4}(1-\xi)(1-\eta)y_1 + \frac{1}{4}(1-\xi)(1+\eta)y_2 \\ &\quad + \frac{1}{4}(1+\xi)(1+\eta)y_3 + \frac{1}{4}(1+\xi)(1-\eta)y_4 \end{aligned} \quad (3.51)$$

and since the Jacobian matrix is given by

$$[\mathbf{J}] = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \quad (3.52)$$



**Figure 3.10** Integration schemes for (a) quadrilateral element with `nip=4` and (b) triangular element with `nip=1`

it is clear that its terms can be obtained from (3.51), once the derivatives of the shape functions with respect to the local coordinates have been provided by subroutine `shape_der`, thus

```
CALL shape_der(der,points,i)
jac = MATMUL(der,coord)
det=determinant(jac)                                (3.53)
```

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

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

```
CALL invert(jac)
deriv = MATMUL(jac,der)                            (3.54)
```

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

The matrix  $[\mathbf{B}]$  in (3.46) (called `bee` in program terminology) can now be assembled as it consists of components of `deriv`, and this operation is performed by the call

```
CALL beemat(bee,deriv)                            (3.55)
```

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

The multiplication

```
btdb=MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)    (3.56)
```

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

$$km = \sum_{i=1}^{nip} det_i * weights(i) * btdb_i (3.57)$$

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

As soon as the element matrix has been formed from (3.57), it can be assembled into the global system matrix (or matrices) by special subroutines described later in this chapter.

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

```
eps = MATMUL(bee,eld)                            (3.58)
```

where, in the case of a 4-node quadrilateral

$$\text{eld} = [u_1 \ v_1 \ u_2 \ v_2 \ u_3 \ v_3 \ u_4 \ v_4]^T \quad (3.59)$$

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

$$\text{sigma} = \text{MATMUL}(\text{dee}, \text{eps}) \quad (3.60)$$

The variables  $u$  and  $v$  are simply the nodal displacements in the  $x$ - and  $y$ -directions, respectively assuming the nodal ordering of Figure 3.10(a).

In cases where the stiffness matrix `km` of a 4-node quadrilateral is required ‘analytically’, the integration loop in Figure 3.9 is replaced by a single call to the subroutine `stiff4` (see, e.g., Program 11.6). Similarly, when strains and stresses are back-calculated using 8-node quadrilateral elements (usually at the sampling points), the ‘analytical’ subroutine `bee8` can be used to replace the lines of program given by (3.53)–(3.55) (see, e.g., Program 6.4).

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

Before describing the assembly process, which is common to most programs in this text, modifications to the element matrix calculation for different situations will first be described.

### 3.7.4 Plane Elastic Analysis using Triangular Elements

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

$$\text{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} \quad (3.61)$$

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

$$\text{fun} = \left\{ \begin{array}{c} L_1 \\ L_3 \\ L_2 \end{array} \right\} \quad (3.62)$$

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

$$\text{der} = \begin{bmatrix} 1 & -1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \quad (3.63)$$

The nodal numbering is shown in Figure 3.10(b). Exactly the same sequence of operations (3.53)–(3.55) as was used for quadrilaterals places the required derivatives with

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

### 3.7.5 Axisymmetric Strain of Elastic Solids

The formation of the strain–displacement matrix follows a similar course to that described by (3.53)–(3.55), however in this case `bee` must be augmented by a fourth row corresponding to the ‘hoop’ strain  $\epsilon_\theta$  as shown in (2.80). The cylindrical coordinates  $(r, z)$  replace their counterparts  $(x, y)$ . The stress–strain matrix is given by equation (2.81) and is returned by subroutine `deemat` with `nst`, the number of stress and strain components now set to 4.

In this case, the integrated element stiffness is given by (2.78), namely

$$[\mathbf{k}_m] = \iint [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] r dr dz \quad (3.64)$$

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

$$\text{gc} = \text{MATMUL}(\text{coord}, \text{fun}) \quad (3.65)$$

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

The numerical integration summation in axisymmetry is written as

$$km = \sum_{i=1}^{nip} \det_i * \text{weights}(i) * \text{btdb}_i * \text{gc}(1)_i \quad (3.66)$$

By comparison with (3.57) it may be seen that when evaluated numerically, the algorithms for axisymmetric and plane stiffness formation will be essentially the same, despite the fact that they are algebraically quite different. This is very significant from the points of view of programming effort and program flexibility (e.g., Program 5.1).

However, (3.58) now involves numerical evaluation of integrals involving  $1/r$  [held in the `[B]` matrix, see (2.80)] which do not have simple polynomial representations. Therefore, in contrast to plane problems, it will be more difficult to evaluate (3.64) by numerical means, and accuracy may deteriorate as  $r$  approaches zero. Provided integration points do not lie on the  $r = 0$  axis, however, reasonable results are usually achieved using a similar order of quadrature to that used in plane analysis. Customised numerical integration schemes for axisymmetric elements are available (Griffiths, 1991), but are not used in this text.

### 3.7.6 Plane Steady Laminar Fluid Flow

It was shown in (2.130) that a fluid element has a ‘stiffness’ or conductivity matrix defined in 2D by

$$[\mathbf{k}_c] = \iint [\mathbf{T}]^T [\mathbf{K}] [\mathbf{T}] dx dy \quad (3.67)$$

and the similarity to (3.46) is obvious. The matrix `deriv` simply contains the derivatives of the element shape functions with respect to  $(x, y)$  which were previously needed in the analysis of solids and formed by the sequence (3.53)–(3.54), while the constitutive matrix  $[\mathbf{K}]$  (called `kay` in program terminology) contains the permeability (or conductivity) properties of the element in the form

$$\text{kay} = \begin{bmatrix} k_x & 0 \\ 0 & k_y \end{bmatrix} \quad (3.68)$$

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

$$\begin{aligned} \text{dtkd} &= \text{MATMUL}(\text{MATMUL}(\text{TRANSPOSE}(\text{deriv}), \text{kay}), \text{deriv}) \\ \text{kc} &= \sum_{i=1}^{nip} \det_i * \text{weights}(i) * \text{dtkd}_i \end{aligned} \quad (3.69)$$

By comparison with (3.57) it will be seen that these physically very different problems are likely to require similar solution algorithms.

### 3.7.7 Mass Matrix Formation

The mass matrix was shown in Chapter 2, for example (2.75), to take the general form

$$[\mathbf{m}_m] = \rho \iint [\mathbf{N}]^T [\mathbf{N}] dx dy \quad (3.70)$$

where  $[\mathbf{N}]$  holds the shape functions.

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

$$\begin{aligned} \text{CALL cross\_product(fun, fun, ntn)} \\ \text{mm} &= \sum_{i=1}^{nip} \det_i * \text{weights}(i) * \text{ntn}_i \end{aligned} \quad (3.71)$$

where `cross_product` forms the array `ntn` prior to integration.

In the case of dynamic applications in plane stress or strain of solids (`nodof=2`), because of the arrangement of the displacement vector in (3.59) it is convenient to use a

special subroutine `ecmat` to form the terms of the mass matrix as `ecm` before integration, hence

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

$$\text{mm} = \text{rho} * \sum_{i=1}^{nip} \det_i * \text{weights}(i) * \text{ecm}_i \quad (3.72)$$

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

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

$$\text{mm} = \text{rho} * \text{area} / \text{nod} * [\mathbf{I}] \quad (3.73)$$

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

### 3.7.8 Higher-Order 2D Elements

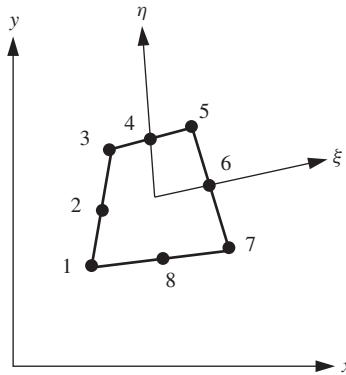
The shape functions and derivatives provided by subroutines `shape_fun` and `shape_der` allow analyses to be performed using quadrilateral elements with 4, 8 or 9 nodes, or triangular elements with 3, 6, 10 or 15 nodes (e.g., Program 5.1). A summary of the shape functions for those elements is given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

#### 8-Node Quadrilateral

To emphasise the ease with which element types can be interchanged in programs, consider the next member of the isoparametric quadrilateral group, namely the 8-node quadratic (‘serendipity’) quadrilateral element with mid-side nodes shown in Figure 3.11.

The coordinate matrix becomes

$$\text{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_5 & y_5 \\ x_6 & y_6 \\ x_7 & y_7 \\ x_8 & y_8 \end{bmatrix} \quad (3.74)$$



**Figure 3.11** General quadratic quadrilateral element

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

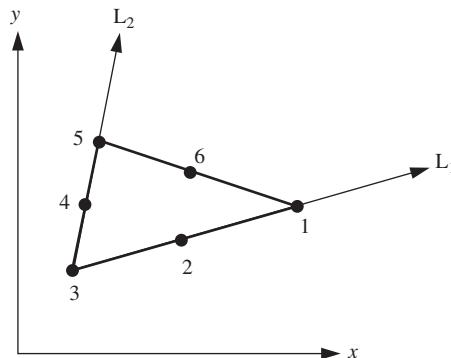
$$\text{fun} = \left\{ \begin{array}{l} \frac{1}{4}(1-\xi)(1-\eta)(-\xi-\eta-1) \\ \frac{1}{2}(1-\xi)(1-\eta^2) \\ \frac{1}{4}(1-\xi)(1+\eta)(-\xi+\eta-1) \\ \frac{1}{2}(1-\xi^2)(1+\eta) \\ \frac{1}{4}(1+\xi)(1+\eta)(\xi+\eta-1) \\ \frac{1}{2}(1+\xi)(1-\eta^2) \\ \frac{1}{4}(1+\xi)(1-\eta)(\xi-\eta-1) \\ \frac{1}{2}(1-\xi^2)(1-\eta) \end{array} \right\} \quad (3.75)$$

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

The sequence of operations described by (3.53)–(3.57) obtains the terms needed for the element stiffness matrix integration.

### 6-Node Triangle

Another plane element available for use with the programs later in this book is the next member of the triangle family, namely the 6-node triangular (`nod=6`) element as shown in Figure 3.12.



**Figure 3.12** General 6-node triangular element

The coordinate matrix becomes

$$\text{coord} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_5 & y_5 \\ x_6 & y_6 \end{bmatrix} \quad (3.76)$$

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

$$\text{fun} = \begin{Bmatrix} (2L_1 - 1)L_1 \\ 4L_3L_1 \\ (2L_3 - 1)L_3 \\ 4L_2L_3 \\ (2L_2 - 1)L_2 \\ 4L_1L_2 \end{Bmatrix} \quad (3.77)$$

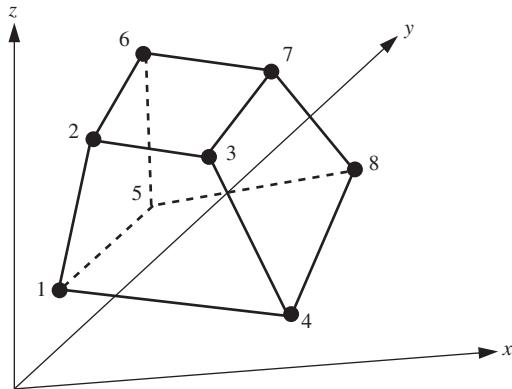
Both the shape functions `fun` and the derivatives `der` are formed as usual by the subroutines `shape_fun` and `shape_der`. The sequence of operations described by (3.53)–(3.57) again follow to generate the stiffness matrix of the element.

### 3.7.9 Three-Dimensional Elements

#### Hexahedral Elements

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

As was the case with changes of plane element types, changes of element dimensions are readily made. For example, the 8-node hexahedral ‘brick’ element in Figure 3.13 is the three-dimensional extension of the 4-node quadrilateral.



**Figure 3.13** General linear hexahedron ‘brick’ element

The coordinate matrix becomes

$$\text{coord} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \\ x_7 & y_7 & z_7 \\ x_8 & y_8 & z_8 \end{bmatrix} \quad (3.78)$$

and using the three-dimensional local coordinate system  $(\xi, \eta, \zeta)$ , the shape functions become

$$\text{fun} = \left\{ \begin{array}{l} \frac{1}{8}(1-\xi)(1-\eta)(1-\zeta) \\ \frac{1}{8}(1-\xi)(1-\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1-\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1-\eta)(1-\zeta) \\ \frac{1}{8}(1-\xi)(1+\eta)(1-\zeta) \\ \frac{1}{8}(1-\xi)(1+\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1+\eta)(1+\zeta) \\ \frac{1}{8}(1+\xi)(1+\eta)(1-\zeta) \end{array} \right\} \quad (3.79)$$

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

The sequence of operations described by (3.53)–(3.54) results in `deriv`, the required gradients with respect to  $(x, y, z)$  and the Jacobian determinant `det`.

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

$$[\mathbf{k}_m] = \iiint [\mathbf{B}]^T [\mathbf{D}] [\mathbf{B}] dx dy dz \quad (3.80)$$

where  $\mathbf{B}$  and  $\mathbf{D}$  are formed by the subroutines `beemat` and `deemat` as usual but with `nst`, the number of components of stress and strain now 6.

The numerical integration summation follows the same course as described previously for 2D elements in equations (3.57).

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

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

$$[\mathbf{k}_c] = \iiint [\mathbf{T}]^T [\mathbf{K}] [\mathbf{T}] dx dy dz \quad (3.81)$$

where the property matrix is formed as

$$[\mathbf{K}] = \begin{bmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{bmatrix} \quad (3.82)$$

Similarly, the ‘mass’ matrix for fluid flow is

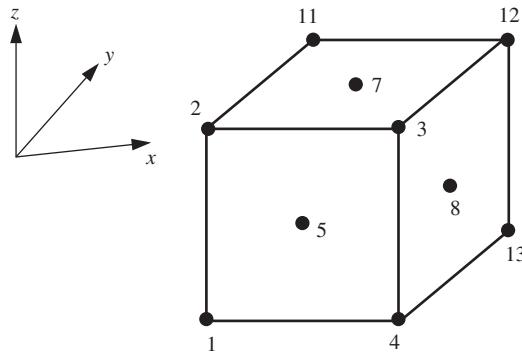
$$[\mathbf{m}_m] = \iiint [\mathbf{N}]^T [\mathbf{N}] dx dy dz \quad (3.83)$$

In both cases, an identical sequence of operations as described previously for planar flow in equations (3.69) and (3.71) delivers the numerically integrated conductivity matrix  $\mathbf{k}_c$  and ‘mass’ matrix  $\mathbf{m}_m$ .

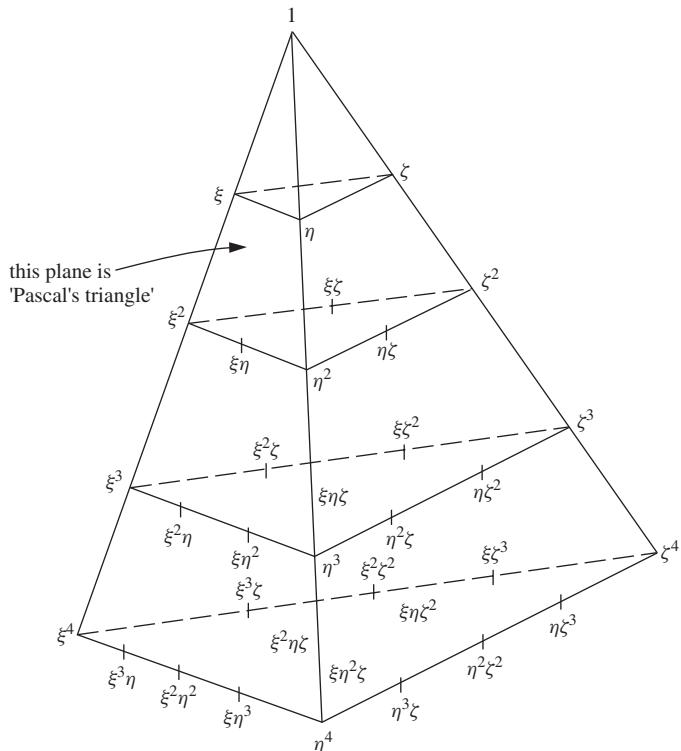
### 14-Node Hexahedral Element

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

An alternative is to use a 14-node element (Smith and Kidger, 1992). As shown in Figure 3.14, this has 8 corner and 6 mid-face nodes. These nodes ‘populate’ three-dimensional space more uniformly than 20 nodes do, since the latter are concentrated along the mesh lines. However, there is no unique choice of shape functions for a 14-node element. Figure 3.15 shows the ‘Pascal pyramid’ of polynomials in  $(\xi, \eta, \zeta)$  (or  $L_1, L_2, L_3$ ) and one could experiment with various combinations of terms. It should



**Figure 3.14** A 14-node hexahedron ‘brick’ element



**Figure 3.15** The Pascal pyramid

be noted that the nearest plane of the pyramid is ‘Pascal’s triangle’, which can be used to select shape function terms for 2D elements. Smith and Kidger (1992) tried six permutations which were called ‘Types 1 to 6’. For example, Type 1 contained all 10 polynomials down to the second ‘plane’ of the pyramid plus the terms  $\xi\eta\zeta$ ,  $\xi^2\eta$ ,  $\eta^2\zeta$  and  $\zeta^2\xi$  from the third ‘plane’. Type 6 selectively contained terms as far down as the fifth ‘plane’ and this

is the version available in library subroutines `shape_fun` and `shape_der`. Computer algebra was essential when deriving the shape functions for the ‘Type 6’ element, which are listed in Appendix B.

The best means of appreciating an element’s properties lie in determination of the eigenvalues and eigenvectors of its ‘stiffness’ matrix, for example  $[\mathbf{k}_m]$ . For a 3D elastic solid element this matrix must possess exactly six zero eigenvalues (associated with the ‘rigid-body’ modes of three translations and rotations). The remaining eigenmodes specify the limited shapes into which the element can deform (Kidger and Smith, 1992). Figure 3.16 illustrates this for the 8-node hexahedron.

### Tetrahedral Elements

An alternative element for 3D analysis is the tetrahedron, the simplest of which has four corner nodes and is called the ‘constant-strain’ tetrahedron. The local coordinate system involves mapping a general tetrahedron onto a right-angled tetrahedron with three orthogonal sides of unit length as shown in Figure 3.17. This approach can be shown to be identical to ‘volume coordinates’. For example, a general point P can be identified uniquely by the coordinates  $(L_1, L_2, L_3)$ . As with triangles, an additional coordinate  $L_4$  given by

$$L_4 = 1 - L_1 - L_2 - L_3 \quad (3.84)$$

is sometimes retained for algebraic convenience.

The shape functions for the ‘constant-strain’ tetrahedron are

$$\text{fun} = \begin{Bmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \end{Bmatrix} \quad (3.85)$$

and these, together with their derivatives with respect to  $L_1$ ,  $L_2$  and  $L_3$  are formed by the usual subroutines `shape_fun` and `shape_der` with `ndim=3` and `nod=4`. The sequence of operations described by (3.53)–(3.57) again follows to generate the stiffness matrix of the element.

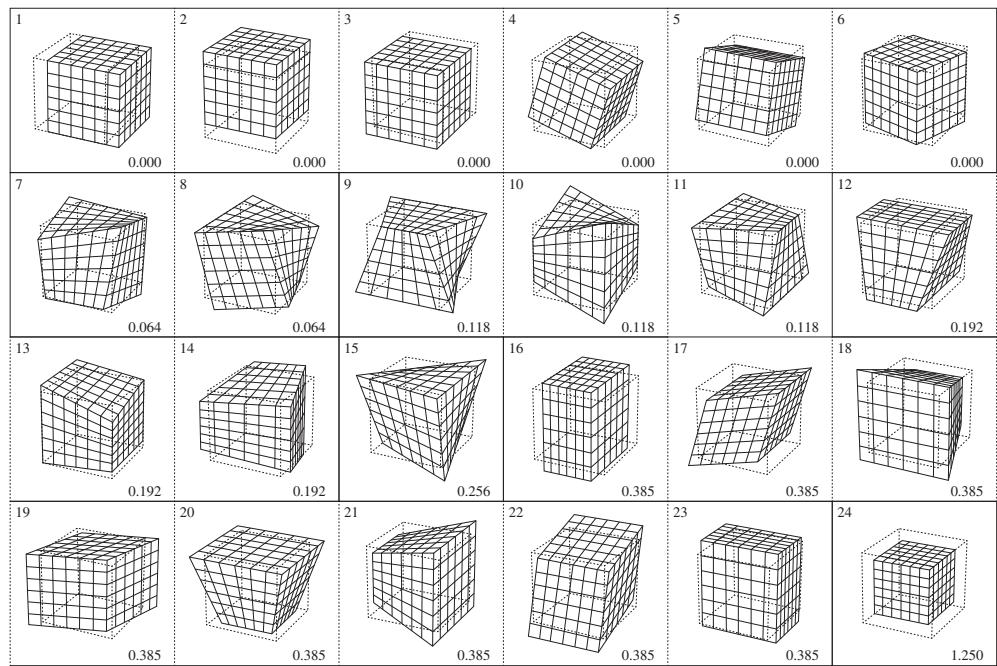
The ‘constant-strain’ tetrahedron requires only one integrating point (`nip=1`) situated at the element centroid.

The addition of mid-side nodes results in the 10-node tetrahedron which represents the next member of the family. Reference to Figure 3.15 and replacing  $(\xi, \eta, \zeta)$  by  $(L_1, L_2, L_3)$  shows that the tetrahedral family of shape functions maps naturally onto the Pascal pyramid (4, 10, 20, 35, etc. nodes). These elements could easily be implemented by the interested reader.

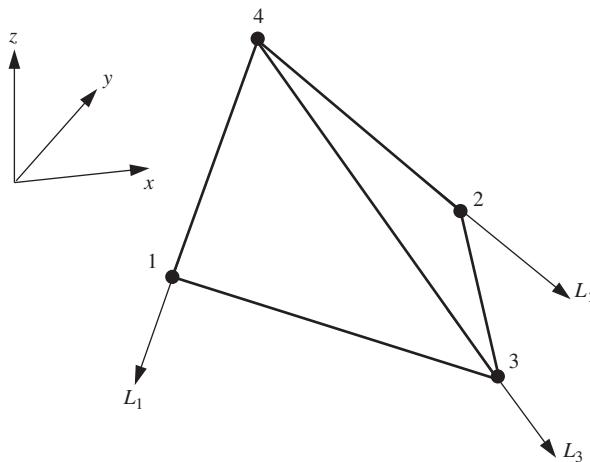
Transient, coupled poroelastic transient and elastic–plastic analysis all involve manipulations of the few simple element property matrices described above. Before describing such applications, methods of assembling elements and of solving linear equilibrium and eigenvalue problems will first be discussed.

#### 3.7.10 Assembly of Elements

The special purpose subroutines `formnf`, `formkb`, `formku`, `fkdiag`, `formtb` and `fsparv` are concerned with assembling the individual element matrices to form the global



**Figure 3.16** Mode shapes and eigenvalues for the 8-node brick: eight integration points;  $\nu = 0.3$



**Figure 3.17** A 4-node tetrahedron element

matrices that approximate the desired continuum, if assembly is preferred to an ‘element-by-element’ approach. Allied to these there must be a specification of the geometrical details, in particular the nodal coordinates of each element and the element’s place in some overall node numbering scheme.

Large finite element programs contain mesh generation code which is usually of some considerable complexity. Indeed, in much finite element work, the most expensive and time-consuming task is the preparation of the input data using the mesh generation routines. In the present book, most examples are restricted to simple classes of geometry, such as those shown in Figure 3.18, which can be automatically built up by ‘geometry’ subroutines.

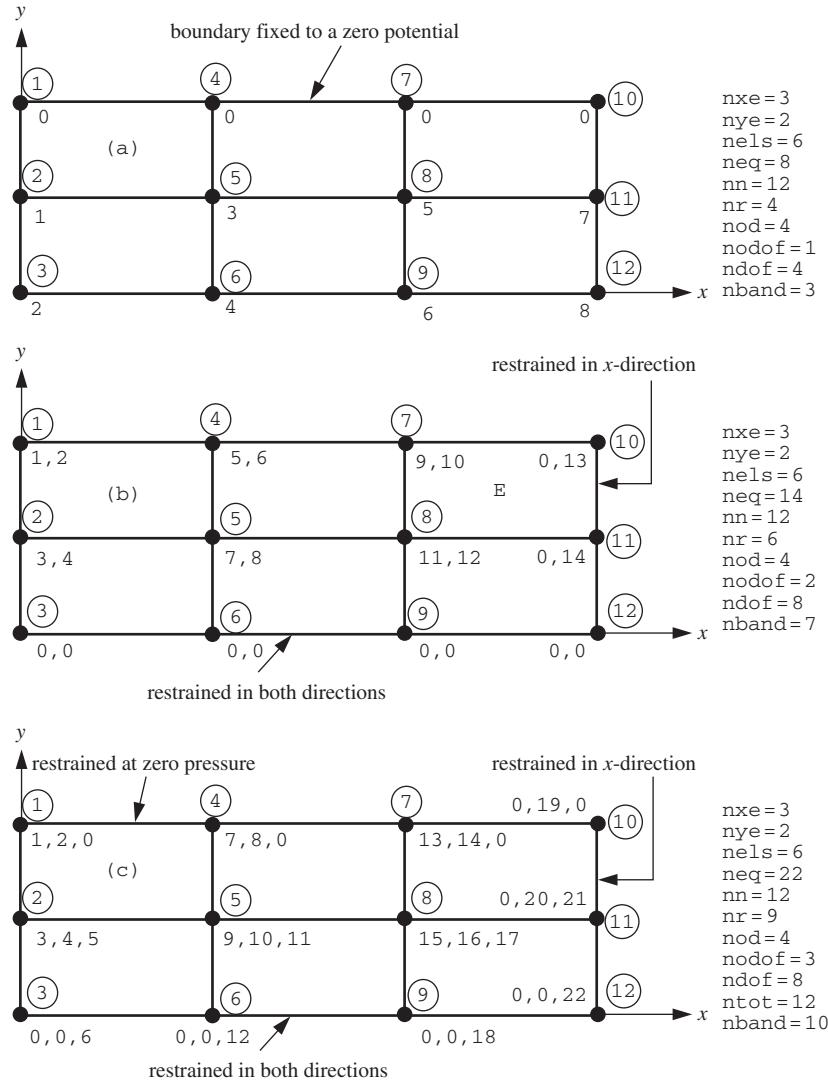
Alternatively, more general purpose programs are presented later in which the element geometries and nodal connectivities are simply read into the analysis program as data, having previously been worked out by an independent mesh generator.

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

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

With reference to Figure 3.18, the nodes of the mesh are first assigned numbers as economically as possible (i.e., always numbering in the ‘shorter’ direction to minimise the bandwidth). Associated with each node are degrees of freedom (displacements, fluid potentials and so on) which are numbered in the same order as the nodes. However, account can be taken at this stage of whether a degree of freedom exists or whether, generally at the boundaries of the region, the freedom is suppressed, in which case that freedom number is assigned the value zero. Alternatively, all freedoms can be assigned values whether they equal zero or not, and fixed later to their required values using the ‘penalty’ approach. This latter approach leads to larger systems of equations, but with simpler freedom numbering.

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



**Figure 3.18** Numbering system and data for regular meshes. (a) One degree of freedom per node. (b) Two degrees of freedom per node. (c) Coupled problem with three degrees of freedom per node

The variables in Figure 3.18 have the following meaning:

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

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

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

In scalar potential problems, there is one degree of freedom possible per node, the ‘potential’  $\phi$  [Figure 3.18(a)]. In plane or axisymmetric strain problems there are two, namely  $u$  and  $v$ , the components of displacement in the  $x$ - and  $y$ - (or  $r$ - and  $z$ -) directions, respectively [Figure 3.18(b)]. In planar coupled solid–fluid problems there are three, with  $u$ ,  $v$  and  $u_w$  [where  $u_w$  = excess pressure, Figure 3.18(c)], and similarly in Navier–Stokes applications the order is  $u$ ,  $p$ ,  $v$  (where  $p$  = pressure, and  $u$  and  $v$  represent velocity components). Regular 3D displacement problems have three freedoms per node given by  $u$ ,  $v$  and  $w$  (in  $x$ ,  $y$  and  $z$ ). For 3D coupled solid–fluid problems, the four degrees of freedom per node are  $u$ ,  $v$ ,  $w$  and  $u_w$ , while for 3D Navier–Stokes the order is  $u$ ,  $p$ ,  $v$  and  $w$ .

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

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

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

For example, to create `nf` for the problem shown in Figure 3.18(b), the data specified and the resulting `nf` are listed in Table 3.6.

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

**Table 3.6** Formation of a typical nodal freedom array

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

**Table 3.7** Summary of assembly subroutines

Subroutine	Banding/Symmetry/ Triangle?	Storage	Diagonals
fspdrv	Yes/Yes/Lower	kv(1:kdiag(neq))	kdiag ‘skyline’
formku	Yes/Yes/Upper	ku(neq,nband+1))	1st col.
formkb	Yes/Yes/Lower	kb(neq,nband+1))	nband+1th col.
formtb	Yes/No/Both	pb(neq,2(nband+1)-1)	nband+1th col.

nodal coordinates (held in array `coord`) and the nodal numbering (held in vector `num`). Both the coordinates and the node numbering are generated in an order consistent with the local node numbering of the element. In the case of a 4-node quadrilateral, this would be in the order 1-2-3-4 as shown in Figure 3.1 (see Appendix B for local numbering of all elements used in this book). For example, element E in Figure 3.18(b) has the node numbering vector

$$\text{num} = [8 \ 7 \ 10 \ 11]^T \quad (3.86)$$

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

$$g = [11 \ 12 \ 9 \ 10 \ 0 \ 13 \ 0 \ 14]^T \quad (3.87)$$

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

A simple three-dimensional mesh is shown in Figure 3.19, and the system coefficients can again be assembled using the same building blocks.

Although the user of these subroutines does not strictly need to know how the storage is carried out, examples are given in Figure 3.20 of the most commonly used storage strategies generated by the subroutines in Table 3.7.

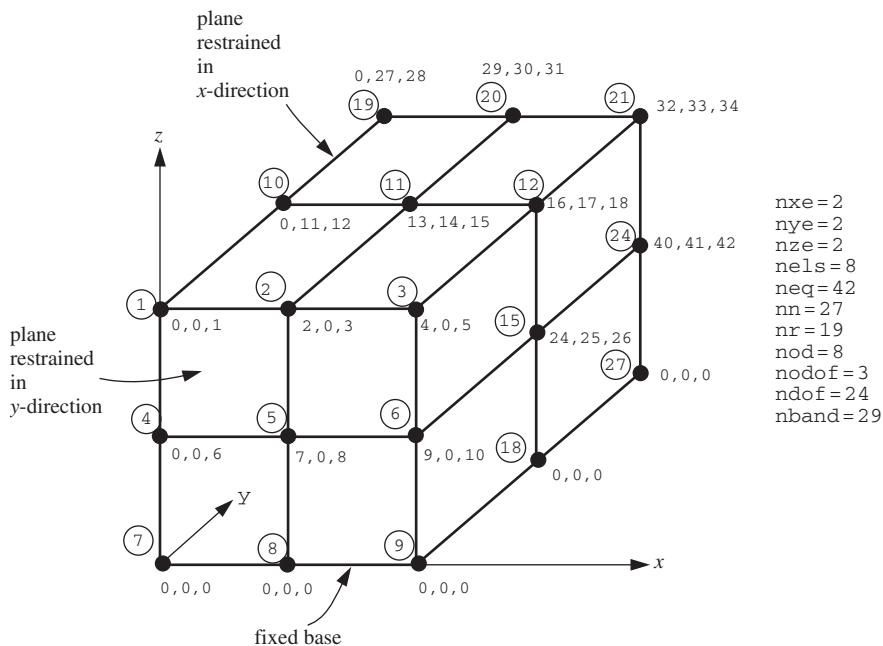
The strategy in the majority of programs in this book is the ‘skyline’ approach. It should be noted that `fspdrv` stores only those numbers within the ‘skyline’ in the form of a vector. Information regarding the position of the diagonal terms within the resulting vector is held in the integer vector `kdiag` formed by subroutine `fkdiag`.

## 3.8 Solution of Equilibrium Equations

If an assembly strategy is chosen, after specification of boundary conditions, a typical global equilibrium equation is

$$[\mathbf{K}_m] \{\mathbf{U}\} = \{\mathbf{F}\} \quad (3.88)$$

in which the terms of the global coefficient matrix  $[\mathbf{K}_m]$  have usually been assembled by subroutine `fspdrv` and stored in a vector called `kv`, and the global right-hand-side vector  $\{\mathbf{F}\}$ , usually stored in a vector called `loads`, is just input as data. The black box subroutines for equation solution for the unknown vector  $\{\mathbf{U}\}$  depend of course on the method of coefficient storage according to the schemes shown in Table 3.8.



**Figure 3.19** Numbering system and data for a regular 3D mesh with three degrees of freedom per node

In fact, to save storage, all of the solution routines overwrite the right-hand side by the solution. That is, following solution of (3.88), the vector  $\text{loads}$  holds the solution. The storage strategy adopted by the compiler and the hardware, as described in Chapter 1, strongly influences the solution method that should be used for large problems.

For very large systems, assembly would not be used at all, and the iterative processes described in Section 3.5 would be substituted.

### 3.9 Evaluation of Eigenvalues and Eigenvectors

Before solution, it is often necessary to reduce the eigenvalue equation to ‘standard form’,

$$[\mathbf{A}] \{\mathbf{Z}\} = \omega^2 \{\mathbf{Z}\} \quad (3.89)$$

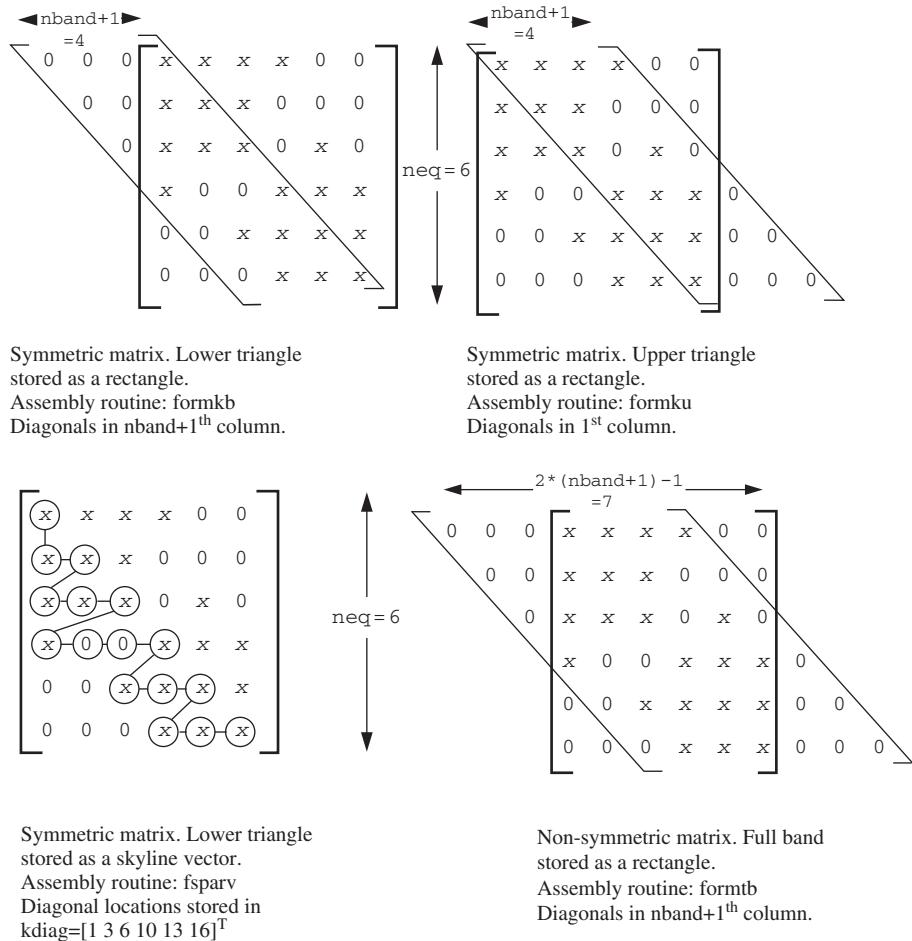
where  $[\mathbf{A}]$  is symmetric,  $\{\mathbf{Z}\}$  represents the eigenvector and  $\omega^2$  the eigenvalue.

#### 3.9.1 Jacobi Algorithm

The problem frequently presents itself in the form of a generalised eigenproblem of the form

$$[\mathbf{K}_m] \{\mathbf{X}\} = \omega^2 [\mathbf{M}_m] \{\mathbf{X}\} \quad (3.90)$$

where  $[\mathbf{K}_m]$  and  $[\mathbf{M}_m]$  are the global stiffness and mass matrices, respectively. For example, the stiffness can be stored as a banded matrix having been formed by subroutine `formku` (Table 3.7). The mass would then be either a banded matrix with the



**Figure 3.20** Examples of storage strategies and assembly subroutines for symmetric and non-symmetric banded arrays ( $neq=6$ ,  $nband=3$ )

**Table 3.8** Equation solution subroutines

Coefficients formed by	Solution routines	Method
fspary	{ sparin spabac	Cholesky
fspary	{ sparin_gauss spabac_gauss	Gauss
formtb	{ gauss_band solve_band	Gauss

same structure as the stiffness or more frequently if the mass is assumed to be ‘lumped’, a diagonal matrix which can be stored in a vector `diag` with the help of subroutine `formlump`. In order to reduce (3.90) to the required form (3.89), it is necessary to factorise the mass matrix by forming

$$[\mathbf{M}_m] = [\mathbf{L}] [\mathbf{L}]^T \quad (3.91)$$

While this is essentially a Cholesky factorisation, it is particularly simple in the case of a diagonal matrix, in which case the diagonal terms in  $[\mathbf{L}]$  are simply the square roots of the diagonal terms in  $[\mathbf{M}_m]$  and the inverse of  $[\mathbf{L}]$  merely consists of the reciprocals of these square roots. In any case,

$$[\mathbf{K}_m] \{\mathbf{X}\} = \omega^2 [\mathbf{L}] [\mathbf{L}]^T \{\mathbf{X}\} \quad (3.92)$$

which is then reduced to standard form by making the substitution

$$[\mathbf{L}]^T \{\mathbf{X}\} = \{\mathbf{Z}\} \quad (3.93)$$

Then

$$[\mathbf{L}]^{-1} [\mathbf{K}_m] [\mathbf{L}]^{-T} \{\mathbf{Z}\} = \omega^2 \{\mathbf{Z}\} \quad (3.94)$$

is of the desired form (3.89). Having solved for  $\{\mathbf{Z}\}$ , the required eigenvectors  $\{\mathbf{X}\}$  are readily recovered using (3.93). The subroutines `bandred` and `bisect` deliver the appropriate eigenvalues.

### 3.9.2 Lanczos and Arnoldi Algorithms

The transformation technique previously described is robust in that it will not fail to find an eigenvalue or to detect multiple roots. However, it is expensive to use on large problems for which, in general, vector iteration methods are preferable. Since the heart of all these involves a matrix-by-vector product as shown in Section 3.5.5, they are ideal for ‘element-by-element’ manipulation, in which case the lumped mass matrix is best used. Program 10.3 in Chapter 10 and Program 12.8 in Chapter 12 use the HSL (2011) package EA25, which implements the work of Parlett and Reid (1981) on the Lanczos algorithm. These subroutines calculate the eigenvalues and eigenvectors of a symmetric matrix, say  $[\mathbf{A}]$  from (3.89), requiring the user only to compute matrix–vector products and whole-vector additions of the form  $[\mathbf{A}] \{\mathbf{V}\} + \{\mathbf{U}\}$  similar to those described in (3.30) for  $\{\mathbf{U}\}$  and  $\{\mathbf{V}\}$  provided by the subroutines. These operations can be carried out element-wise, as was described in Section 3.5.

There is a slight additional complexity in that, as in (3.90), we often have the generalised eigenvalue problem to solve. For consistent mass approximations,  $[\mathbf{L}][\mathbf{L}]^T$  in (3.91) is formed by Cholesky factorisation using `chol`. Then on each Lanczos step from (3.30), wherever  $[\mathbf{L}]^{-1} [\mathbf{K}_m] [\mathbf{L}]^{-T} \{\mathbf{V}\} + \{\mathbf{U}\}$  is called for, we compute

$[\mathbf{L}]^T \{\mathbf{Z}_1\} = \{\mathbf{V}\}$	<code>chobk2</code>	(3.95)
$\{\mathbf{Z}_2\} = [\mathbf{K}_m] \{\mathbf{Z}_1\}$	<code>banmul</code>	
$[\mathbf{L}] \{\mathbf{Z}_3\} = \{\mathbf{Z}_2\}$	<code>chobk1</code>	
$\{\mathbf{U}\} = \{\mathbf{U}\} + \{\mathbf{Z}_3\}$	whole-vector operation	

Finally, the true eigenvectors are recovered from the transformed ones using backward substitution [chobk2 as in (3.93)]. For lumped mass approximations, essentially the same operations are performed but the diagonal nature of  $[\mathbf{L}]$  means that subroutine calls can be dispensed with. As an alternative to Lanczos, Arnoldi's method as implemented in the freely available package ARPACK can be used (see Program 10.4). Programs using either of these methods will be seen to be almost identical, with the exception of two subroutine calls.

### 3.10 Solution of First-Order Time-Dependent Problems

A typical equation at the element level from (2.138) is given by

$$[\mathbf{k}_c]\{\phi\} + [\mathbf{m}_m] \left\{ \frac{d\phi}{dt} \right\} = \{\mathbf{q}\} \quad (3.96)$$

where  $\{\phi\}$  represents the dependent variable, and  $\{\mathbf{q}\}$  represents any additional sources or sinks, and may be a function of time. There are many ways of integrating this set of ordinary differential equations, and modern methods for small numbers of equations would probably be based on variable order, variable time step methods with error control. However, for large engineering systems more primitive methods are still mainly used, involving linear interpolations and fixed time steps  $\Delta t$ . If an element assembly method is to be used,  $[\mathbf{k}_c]$  becomes  $[\mathbf{K}_c]$ ,  $[\mathbf{m}_m]$  becomes  $[\mathbf{M}_m]$  and the basic equations can be written at two consecutive time steps '0' and '1' as follows:

$$[\mathbf{K}_c]\{\Phi\}_0 + [\mathbf{M}_m] \left\{ \frac{d\Phi}{dt} \right\}_0 = \{\mathbf{Q}\}_0 \quad (3.97)$$

$$[\mathbf{K}_c]\{\Phi\}_1 + [\mathbf{M}_m] \left\{ \frac{d\Phi}{dt} \right\}_1 = \{\mathbf{Q}\}_1 \quad (3.98)$$

where  $\{\Phi\}$  and  $\{\mathbf{Q}\}$  represent the global counterparts of  $\{\phi\}$  and  $\{\mathbf{q}\}$ .

A third equation advances the solution from '0' to '1' using a weighted average of the gradients at the beginning and end of the time interval, thus

$$\{\Phi\}_1 = \{\Phi\}_0 + \Delta t \left( (1 - \theta) \left\{ \frac{d\Phi}{dt} \right\}_0 + \theta \left\{ \frac{d\Phi}{dt} \right\}_1 \right) \quad (3.99)$$

Elimination of  $\{d\Phi/dt\}_0$  and  $\{d\Phi/dt\}_1$  from equations (3.97)–(3.99) leads to the following recurrence equation between time steps '0' and '1':

$$\begin{aligned} ([\mathbf{M}_m] + \theta \Delta t [\mathbf{K}_c])\{\Phi\}_1 &= ([\mathbf{M}_m] - (1 - \theta) \Delta t [\mathbf{K}_c])\{\Phi\}_0 \\ &\quad + \theta \Delta t \{\mathbf{Q}\}_1 + (1 - \theta) \Delta t \{\mathbf{Q}\}_0 \end{aligned} \quad (3.100)$$

This system is only unconditionally 'stable' (i.e., errors will not grow unboundedly) if  $\theta \geq 1/2$ . Common choices would be  $\theta = 1/2$ , giving the 'Crank–Nicolson' method, where, assuming for the moment  $\{\mathbf{Q}\} = \{\mathbf{0}\}$ ,

$$\left( [\mathbf{M}_m] + \frac{\Delta t}{2} [\mathbf{K}_c] \right) \{\Phi\}_1 = \left( [\mathbf{M}_m] - \frac{\Delta t}{2} [\mathbf{K}_c] \right) \{\Phi\}_0 \quad (3.101)$$

and  $\theta = 1$  giving the ‘fully implicit’ method

$$([\mathbf{M}_m] + \Delta t [\mathbf{K}_c])\{\Phi\}_1 = [\mathbf{M}_m]\{\Phi\}_0 \quad (3.102)$$

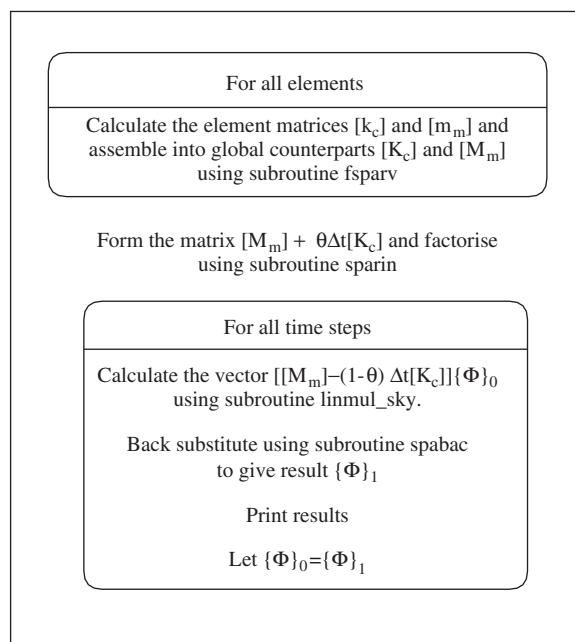
In programming terms, the marching process from ‘0’ to ‘1’ involves first a matrix-by-vector multiplication on the right-hand side of (3.100) or (3.101), using `linmul_sky` (assuming subroutine `fsparv` has been used for assembly), followed by the addition of two vectors if sources are present, and second, a set of linear equations must be solved on each time step. If the  $[\mathbf{K}_c]$  and  $[\mathbf{M}_m]$  matrices do not change with time, the factorisation of the left-hand-side coefficients, which is a time-consuming operation, need only be performed once, as shown in the structure chart in Figure 3.21.

The ‘implicit’ strategies described above are quite effective for linear problems (constant  $[\mathbf{K}_c]$  and  $[\mathbf{M}_m]$ ), however storage requirements can be considerable, and in non-linear problems the necessity to refactorise  $([\mathbf{M}_m] + \theta \Delta t [\mathbf{K}_c])$  can lead to lengthy calculations.

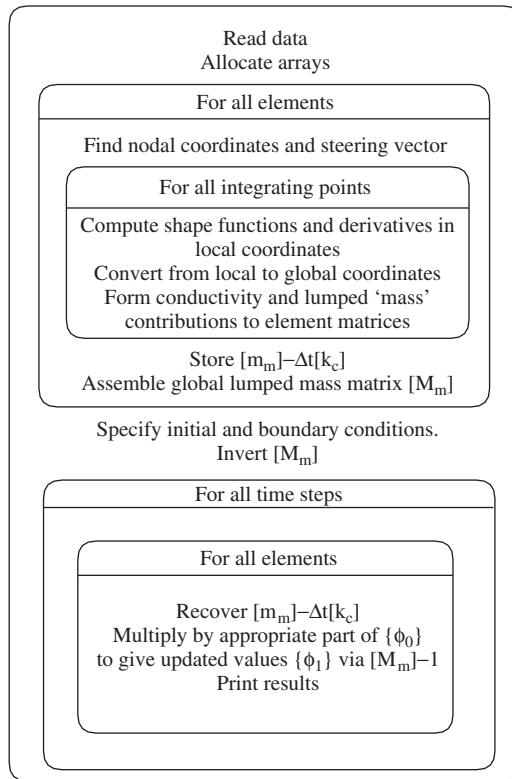
Storage can be saved by replacing subroutine `linmul_sky` by an ‘element-by-element’ matrix–vector multiply, and by solving the simultaneous equations iteratively [by the preconditioned conjugate gradient (pcg) method, for example]. Such a strategy can be attractive for parallel processing, and examples are given in Chapter 12.

There is another alternative, widely used for second-order problems (see also Section 3.13.5), in which  $\theta$  is set to zero and the  $[\mathbf{M}_m]$  matrix is ‘lumped’ [see equation (3.73)]. In this ‘explicit’ approach, the system to be solved is

$$[\mathbf{M}_m]\{\Phi\}_1 = ([\mathbf{M}_m] - \Delta t [\mathbf{K}_c])\{\Phi\}_0 \quad (3.103)$$



**Figure 3.21** Structure chart for first-order time-dependent problems by implicit methods using an assembly strategy



**Figure 3.22** Structure chart for first-order time-dependent problems by an explicit method without assembly

or

$$\{\Phi\}_1 = [\mathbf{M}_m]^{-1}([\mathbf{M}_m] - \Delta t[\mathbf{K}_c])\{\Phi\}_0 \quad (3.104)$$

Although written here at the global level, in the case of  $\theta = 0$  no global matrix assembly is needed because the matrix–vector products on the right-hand side of equation (3.104) can all be achieved using an ‘element-by-element’ strategy involving manipulations of the element matrices  $[\mathbf{k}_c]$  and  $[\mathbf{m}_m]$ .

Although the ‘explicit’ algorithm is simple, the disadvantage is that (3.104) is only stable on condition that  $\Delta t$  is ‘small’, and in practice perhaps so small that real times of interest would require an excessive number of steps. A typical structure chart for an ‘explicit’ algorithm is given in Figure 3.22.

Yet another ‘element-by-element’ approach which conserves computer storage while preserving the stability properties of ‘implicit methods’ involves ‘operator splitting’ on an ‘element-by-element’ product basis (Hughes *et al.*, 1983; Smith *et al.*, 1989). Although not necessary for the operation of the method, the simplest algorithms result from ‘lumping’  $[\mathbf{M}_m]$ . Assuming again that  $\{\mathbf{Q}\} = \{0\}$ , equation (3.100) can be written

$$\{\Phi\}_1 = ([\mathbf{M}_m] + \theta \Delta t [\mathbf{K}_c])^{-1}([\mathbf{M}_m] - (1 - \theta) \Delta t [\mathbf{K}_c])\{\Phi\}_0 \quad (3.105)$$

The ‘element-by-element operator splitting’ methods are based on binomial theorem expansions of  $([\mathbf{M}_m] + \theta \Delta t [\mathbf{K}_c])^{-1}$  which neglect product terms. When  $[\mathbf{M}_m]$  is ‘lumped’ (diagonal), the method is particularly straightforward because  $[\mathbf{M}_m]$  can effectively be replaced by  $[\mathbf{I}]$ , the unit matrix, where

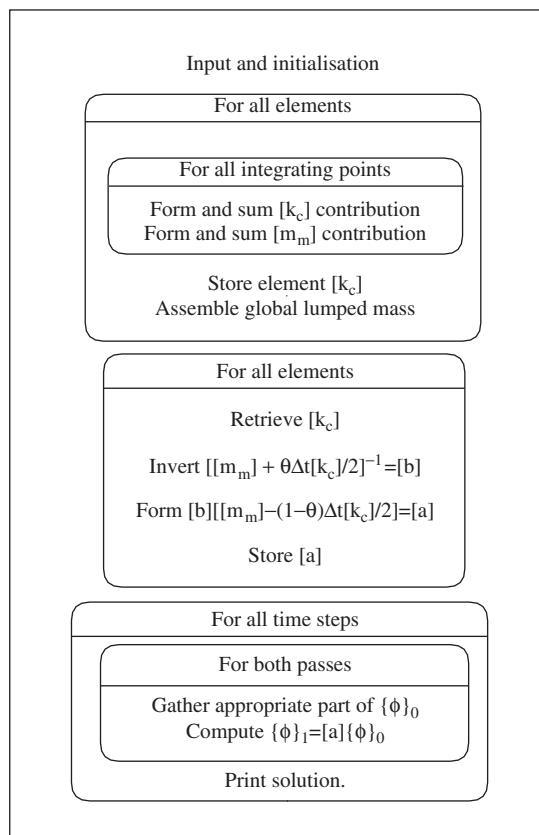
$$([\mathbf{I}] + \theta \Delta t [\mathbf{K}_c])^{-1} = \left( [\mathbf{I}] + \theta \Delta t \sum [\mathbf{k}_c] \right)^{-1} \quad (3.106)$$

$$\approx \prod ([\mathbf{I}] + \theta \Delta t [\mathbf{k}_c])^{-1} \quad (3.107)$$

where  $\sum$  indicates a summation, and  $\prod$  indicates a product over all the elements. As was the case with implicit methods, optimal accuracy consistent with stability is achieved for  $\theta = 1/2$ . It is shown by Hughes *et al.* (1983) that further optimisation is achieved by splitting further to

$$[\mathbf{k}_c] = \frac{1}{2} [\mathbf{k}_c] + \frac{1}{2} [\mathbf{k}_c] \quad (3.108)$$

and carrying out the product (3.107) by sweeping twice through the elements. They suggest from first to last and back again, but clearly various choices of sweeps could be employed. It can be shown that as  $\Delta t \rightarrow 0$ , any of these processes converges to the true solution of the global problem. A structure chart for the process is shown in Figure 3.23,



**Figure 3.23** Structure chart for the ‘element-by-element’ product algorithm (two-pass)

and examples of all the methods described in this section are implemented in Chapter 8. Consistent mass versions are described by Gladwell *et al.* (1989).

### 3.11 Solution of Coupled Navier–Stokes Problems

For steady-state conditions, it was shown in Section 2.16 that a non-linear system of algebraic equations had to be solved, involving, at the element level, submatrices  $[\mathbf{c}_{11}]$ ,  $[\mathbf{c}_{12}]$ , etc. These element matrices contained velocities  $\bar{u}$  and  $\bar{v}$  (called `ubar` and `vbar` in the programs) together with shape functions and their derivatives for the velocity and pressure variables. It was mentioned that it would be possible to use different shape functions for the velocity (vector) quantity and pressure (scalar) quantity, and this is what is done in programs in Chapters 9 and 12.

The velocity shape functions are designated as `fun` and the pressure shape functions as `funf`. Similarly, the velocity derivatives are `deriv` and the pressure derivatives `derivf`. The arrays `nd1`, `nd2`, `ndf1`, `ndf2`, `nfd1` and `nfd2` hold the results of cross-products between the velocity and pressure shape functions and their derivatives as shown below.

Thus, the element integrals which have to be evaluated numerically from equations (2.116)–(2.117) are of the form

```
dtkd=MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)
CALL cross_product(fun,deriv(1,:),nd1)                               (3.109)
CALL cross_product(fun,deriv(2,:),nd2)
```

followed by

$$\begin{aligned} c11 \quad (=c33) = & \sum_{i=1}^{nip} \det_i * \text{weights}(i) * dtkd_i \\ & + \sum_{i=1}^{nip} \text{ubar}_i * \det_i * \text{weights}(i) * nd1_i \\ & + \sum_{i=1}^{nip} \text{vbar}_i * \det_i * \text{weights}(i) * nd2_i \end{aligned} \quad (3.110)$$

In these equations, `deriv(1,:)` signifies the first row of `deriv` and so on in the usual FORTRAN style. Note the identity of the first term of `c11` with (3.69) for uncoupled flow.

The remaining submatrices are formed as follows:

```
CALL cross_product(fun,derivf(1,:),ndf1)
CALL cross_product(fun,derivf(2,:),ndf2)
CALL cross_product(funf,deriv(1,:),nfd1)                               (3.111)
CALL cross_product(funf,deriv(2,:),nfd2)
```

followed by

$$\begin{aligned}
 c12 &= \frac{1}{\text{rho}} \sum_{i=1}^{\text{nip}} \det_i * \text{weights}(i) * \text{ndf1}_i \\
 c32 &= \frac{1}{\text{rho}} \sum_{i=1}^{\text{nip}} \det_i * \text{weights}(i) * \text{ndf2}_i \\
 c21 &= \sum_{i=1}^{\text{nip}} \det_i * \text{weights}(i) * \text{nfd1}_i \\
 c23 &= \sum_{i=1}^{\text{nip}} \det_i * \text{weights}(i) * \text{nfd2}_i
 \end{aligned} \tag{3.112}$$

where rho is the mass density. The other submatrices  $c13$ ,  $c22$  and  $c31$  are set to zero.

The (unsymmetrical) matrix built up from these submatrices is called  $[\mathbf{k}_e]$ , and is formed by a special subroutine called `formupv` (`formupvw` in 3D) and the global, unsymmetrical, band matrix is assembled using `formtb`. The appropriate equation solution routines are `gauss_band` and `solve_band`, as shown in Table 3.8. In an ‘element-by-element’ context, the iterative solution method is `BiCGStab(l)` following the algorithm described in (3.27)–(3.29).

### 3.12 Solution of Coupled Transient Problems

The element equations for Biot consolidation were shown in equation (2.146) to be given by

$$\begin{aligned}
 [\mathbf{k}_m] \{\mathbf{u}\} + [\mathbf{c}] \{\mathbf{u}_w\} &= \{\mathbf{f}\} \\
 [\mathbf{c}]^T \left\{ \frac{d\mathbf{u}}{dt} \right\} - [\mathbf{k}_c] \{\mathbf{u}_w\} &= \{\mathbf{0}\}
 \end{aligned} \tag{3.113}$$

where  $[\mathbf{k}_m]$  and  $[\mathbf{k}_c]$  are the now familiar solid stiffness and fluid conductivity matrices. The matrix  $[\mathbf{c}]$  is the connectivity matrix which is formed from integrals of the form

$$\int \int \frac{\partial N_i}{\partial x} N_j dx dy \tag{3.114}$$

where the first derivative term comes from the displacement field, and the second term comes from the excess pore pressure field. The programs described in Chapter 9 use different element types for the displacements (8-node) and the excess pore pressures (4-node). In Chapter 12, the 3D elements have 20 displacement and 8 pore pressure nodes.

The integrals that generate the  $[\mathbf{c}]$  matrix involve the product of the vectors `vol` and `funf`, where `vol` is derived from the familiar `deriv` array for the solid elements, and takes the form, for 2D,

$$\text{vol} = \left[ \frac{\partial N_1}{\partial x} \quad \frac{\partial N_1}{\partial y} \quad \frac{\partial N_2}{\partial x} \quad \frac{\partial N_2}{\partial y} \quad \dots \quad \dots \quad \frac{\partial N_8}{\partial x} \quad \frac{\partial N_8}{\partial y} \right]^T \tag{3.115}$$

and `funf` holds the fluid component shape functions as

$$\text{funf} = [N_1 \ N_2 \ N_3 \ N_4]^T \quad (3.116)$$

The following sequence completes the integration:

```
CALL cross_product(vol, funf, volf)
c =  $\sum_{i=1}^{nip} \det_i * \text{weights}(i) * \text{volf}_i$  \quad (3.117)
```

The volumetric strain ( $\epsilon_v$ ) at any point within a displacement element is easily retrieved from the product, `DOT_PRODUCT(vol, eld)`, where `eld` holds the element nodal displacements.

To integrate equations (3.113) with respect to time there are again many methods available, but we consider only the simplest linear interpolation in time using finite differences, similar to that used for first-order uncoupled problems in equation (3.100).

### 3.12.1 Absolute Load Version

This approach, used in earlier editions of the book but described here for completeness, applies the full external loading at each time step and is an option for linear elastic problems. Interpolation in time using  $\theta$ , and elimination of the derivative terms as was done for first-order problems in (3.100), leads to the following recurrence equations at the element level:

$$\begin{bmatrix} \theta[\mathbf{k}_m] & \theta[\mathbf{c}] \\ \theta[\mathbf{c}]^T & -\theta^2 \Delta t [\mathbf{k}_c] \end{bmatrix} \begin{Bmatrix} \{\mathbf{u}\} \\ \{\mathbf{u}_w\} \end{Bmatrix}_1 = \begin{bmatrix} -(1-\theta)[\mathbf{k}_m] & -(1-\theta)[\mathbf{c}] \\ \theta[\mathbf{c}]^T & \theta(1-\theta)\Delta t [\mathbf{k}_c] \end{bmatrix} \begin{Bmatrix} \{\mathbf{u}\} \\ \{\mathbf{u}_w\} \end{Bmatrix}_0 + \begin{Bmatrix} (1-\theta)\{\mathbf{f}\} \\ \{\mathbf{0}\} \end{Bmatrix}_0 + \begin{Bmatrix} \theta\{\mathbf{f}\} \\ \{\mathbf{0}\} \end{Bmatrix}_1 \quad (3.118)$$

where  $\{\mathbf{f}\}$  represents the external loading vector which may itself be time-dependent.

Calling the left- and right-hand-side element matrices in (3.118)  $[\mathbf{k}_e]$  and  $[\mathbf{k}_d]$ , respectively, it may be noted that the second equation has been multiplied through by  $\theta$  to preserve symmetry of the  $[\mathbf{k}_e]$  matrix, however the right-hand-side matrix  $[\mathbf{k}_d]$  is unsymmetric.

A Crank–Nicolson type of approximation,  $\theta = 1/2$  would be a popular choice in equation (3.118), however it will be shown in Chapter 8 that this approximation can lead to oscillatory results. The oscillations can be smoothed out either by using the fully implicit version with  $\theta = 1$ , or by writing the first of (3.118) with  $\theta = 1$  and the second with  $\theta = 1/2$ . Examples of this algorithm with constant  $\theta$  are presented in Chapter 9. In all cases, a right-hand-side matrix-by-vector multiplication is followed by an equation solution for each time step. As before, a saving in computer time can be achieved if  $[\mathbf{k}_m]$ ,  $[\mathbf{c}]$  and  $[\mathbf{k}_c]$  are independent of time, and constant  $\Delta t$  is used, because the left-hand-side matrix needs to be factorised only once.

Note that the left-hand-side matrix is always symmetrical whereas the right-hand-side is not. Therefore, if using an assembly approach `fspdrv` can be used to assemble the left-hand-side system equations from (3.118), where `formtb` must be used to assemble the right-hand-side system followed by `bantmul` (Table 3.5) to complete the matrix–vector

multiply. ‘Element-by-element’ summation is most effective in the right-hand-side operations in (3.118) due to sparsity, and ‘element-by-element’ algorithms can be developed using pcg equation solution as shown in Chapters 9 and 12.

### 3.12.2 Incremental Load Version

In (3.113),  $\{\mathbf{f}\}$  is the total force applied and these equations are appropriate to linear systems. Later in this book we shall be concerned with non-linear systems similar to those described in Chapter 6, in which it is desirable to apply loads incrementally and allow plastic stress redistribution to equilibrate at each step.

If  $\{\Delta\mathbf{f}\}$  is the change in load between successive times, the incremental form of the first of (3.113) is

$$[\mathbf{k}_m]\{\Delta\mathbf{u}\} + [\mathbf{c}]\{\Delta\mathbf{u}_w\} = \{\Delta\mathbf{f}\} \quad (3.119)$$

where  $\{\Delta\mathbf{u}\}$  and  $\{\Delta\mathbf{u}_w\}$  are the resulting changes in displacement and excess pore pressure, respectively. Linear interpolation in time using the  $\theta$ -method yields

$$\{\Delta\mathbf{u}\} = \Delta t \left( (1 - \theta) \left\{ \frac{d\mathbf{u}}{dt} \right\}_0 + \theta \left\{ \frac{d\mathbf{u}}{dt} \right\}_1 \right) \quad (3.120)$$

and the second of (3.113) can be written at the two time levels to give expressions for the derivatives, which can then be eliminated to give the following incremental recurrence equations (e.g., Sandhu and Wilson, 1969; Griffiths, 1994a; Hicks, 1995):

$$\begin{bmatrix} [\mathbf{k}_m] & [\mathbf{c}] \\ [\mathbf{c}]^T & -\theta \Delta t [\mathbf{k}_c] \end{bmatrix} \begin{Bmatrix} \{\Delta\mathbf{u}\} \\ \{\Delta\mathbf{u}_w\} \end{Bmatrix} = \begin{Bmatrix} \{\Delta\mathbf{f}\} \\ \Delta t [\mathbf{k}_c]\{\mathbf{u}_w\}_0 \end{Bmatrix} \quad (3.121)$$

The left-hand-side element matrix, again called  $[\mathbf{k}_e]$ , is formed from its constituent matrices by subroutine `formke` and is symmetric. If using an assembly strategy, subroutine `fsparv` generates the global matrix. The right-hand-side vector consists of load increments  $\{\Delta\mathbf{f}\}$  and fluid ‘loads’ given by  $\Delta t [\mathbf{k}_c]\{\mathbf{u}_w\}_0$ . The fluid term is conveniently computed without any need for assembly using an ‘element-by-element’ product approach (see Program 9.4). Subroutines `sparin` and `spabac` complete the solution for the incremental displacements and excess pore pressures.

At each time step, all that remains is to update the dependent variables using

$$\begin{aligned} \{\mathbf{u}\}_1 &= \{\mathbf{u}\}_0 + \{\Delta\mathbf{u}\} \\ \{\mathbf{u}_w\}_1 &= \{\mathbf{u}_w\}_0 + \{\Delta\mathbf{u}_w\} \end{aligned} \quad (3.122)$$

It is the incremental version of the coupled transient equations that is programmed in Chapters 9 and 12.

## 3.13 Solution of Second-Order Time-Dependent Problems

The basic second-order propagation type of equation was derived in Chapter 2 and at the element level takes the form of (2.106), namely

$$[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{m}_m] \left\{ \frac{d^2\mathbf{u}}{dt^2} \right\} = \{\mathbf{f}(t)\} \quad (3.123)$$

where in the context of solid mechanics,  $[\mathbf{k}_m]$  is the element elastic stiffness and  $[\mathbf{m}_m]$  the element mass. In addition to these elastic and inertial forces, solids in motion experience a third type of force whose action is to dissipate energy. For example, the solid may deform so much that plastic strains result, or may be subjected to internal or external friction. Although these phenomena are non-linear in character and can be treated by the non-linear analysis techniques given in Chapter 6, it has been common to linearise the dissipative forces, for example by assuming that they are proportional to velocity. This allows (3.123) to be modified to

$$[\mathbf{k}_m]\{\mathbf{u}\} + [\mathbf{c}_m] \left\{ \frac{d\mathbf{u}}{dt} \right\} + [\mathbf{m}_m] \left\{ \frac{d^2\mathbf{u}}{dt^2} \right\} = \{\mathbf{f}(t)\} \quad (3.124)$$

where  $[\mathbf{c}_m]$  is assumed to be a constant element damping matrix.

Although in principle  $[\mathbf{c}_m]$  could be independently measured or assessed, it is common practice to assume that  $[\mathbf{c}_m]$  is taken to be a linear combination of  $[\mathbf{m}_m]$  and  $[\mathbf{k}_m]$ , where

$$[\mathbf{c}_m] = f_m [\mathbf{m}_m] + f_k [\mathbf{k}_m] \quad (3.125)$$

where  $f_m$  and  $f_k$  are scalars, the so-called ‘Rayleigh’ damping coefficients. They can be related to the more usual ‘damping ratio’  $\gamma$  (Timoshenko *et al.*, 1974) by means of

$$\gamma = \frac{f_m + f_k \omega^2}{2\omega} \quad (3.126)$$

where  $\omega$  is the natural (usually fundamental) frequency of vibration.

The most generally applicable technique for integrating (3.124) with respect to time is ‘direct integration’, in an analogous way to that previously described for first-order problems. Two of the simplest popular implicit methods are described in subsequent sections, where the solution is advanced by one time interval  $\Delta t$ , the values of the displacement and its derivatives at one instant in time being sufficient to determine these values at the subsequent instant by means of recurrence relations. Both preserve unconditional stability, and examples that utilise both element assembly and ‘element-by-element’ strategies are presented in Chapters 11 and 12.

Attention is first focused, however, on the ‘modal superposition’ method.

### 3.13.1 Modal Superposition

This method has as its basis the free undamped part of (3.124), that is when  $[\mathbf{c}_m]$  and  $\{\mathbf{f}\}$  are zero. The reduced equation in assembled form is

$$[\mathbf{K}_m]\{\mathbf{U}\} + [\mathbf{M}_m] \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\} = \{\mathbf{0}\} \quad (3.127)$$

which can of course be converted into an eigenproblem by the assumption of harmonic motion

$$\{\mathbf{U}\} = \{\mathbf{A}\} \sin(\omega t + \psi) \quad (3.128)$$

to give

$$[\mathbf{K}_m]\{\mathbf{A}\} - \omega^2[\mathbf{M}_m]\{\mathbf{A}\} = \{\mathbf{0}\} \quad (3.129)$$

Solution of this eigenproblem by the techniques previously described results in `nEq` eigenvalues  $\omega^2$  and eigenvectors  $\{\mathbf{A}\}$ , where `nEq` (in program terminology) is the total number of degrees of freedom in the finite element mesh. These eigenvectors or ‘mode shapes’ can be considered to be columns of a modal matrix  $[\mathbf{P}]$ , where

$$[\mathbf{P}] = \left[ \begin{array}{c} \{\mathbf{A}_1\} \quad \{\mathbf{A}_2\} \quad \dots \quad \{\mathbf{A}_{n\text{modes}}\} \end{array} \right] \quad (3.130)$$

where `nmodes` is the number of modes that are contributing to the time response. Often it is not necessary to include the higher-frequency components in an analysis, so that `nmodes`  $\leq$  `nEq`.

Because of the properties of eigenproblems, mode shapes possess orthogonality, one to the other, such that

$$\left. \begin{array}{l} \{\mathbf{A}_i\}^T [\mathbf{M}_m] \{\mathbf{A}_j\} = 0 \\ \{\mathbf{A}_i\}^T [\mathbf{K}_m] \{\mathbf{A}_j\} = 0 \end{array} \right\} \quad i \neq j \quad (3.131)$$

$$\left. \begin{array}{l} \{\mathbf{A}_i\}^T [\mathbf{M}_m] \{\mathbf{A}_j\} = m'_{ii} \\ \{\mathbf{A}_i\}^T [\mathbf{K}_m] \{\mathbf{A}_j\} = k'_{ii} \end{array} \right\} \quad i \neq j \quad (3.132)$$

where  $m'_{ii}$  and  $k'_{ii}$  are the diagonal terms of the diagonal global ‘principal’ mass and stiffness matrices,  $[\mathbf{M}']$  and  $[\mathbf{K}']$ , respectively. Use of these relationships in (3.131)–(3.132) has the effect of uncoupling the equations in terms of the principal or ‘normal’ coordinates  $\{\mathbf{U}'\}$ , thus

$$[\mathbf{K}'] \{\mathbf{U}'\} + [\mathbf{M}'] \left\{ \frac{d^2 \mathbf{U}'}{dt^2} \right\} = \{\mathbf{0}\} \quad (3.133)$$

The effect of uncoupling has been to reduce the vibration problem to a set of `nmodes`-independent second-order equations (3.133).

The actual displacements can be retrieved from the normal coordinates by a final superposition process given by

$$\{\mathbf{U}\} = [\mathbf{P}] \{\mathbf{U}'\} \quad (3.134)$$

## Inclusion of Damping

Free damped vibrations, governed by

$$[\mathbf{K}_m] \{\mathbf{U}\} + [\mathbf{C}_m] \left\{ \frac{d \mathbf{U}}{dt} \right\} + [\mathbf{M}_m] \left\{ \frac{d^2 \mathbf{U}}{dt^2} \right\} = \{\mathbf{0}\} \quad (3.135)$$

can be handled by the above technique if it is assumed that the undamped mode shapes are also orthogonal with respect to the damping matrix  $[\mathbf{C}_m]$  in the way described by (3.131)–(3.132). This can readily be achieved if  $[\mathbf{C}_m]$  is taken to be a linear combination of  $[\mathbf{M}_m]$  and  $[\mathbf{K}_m]$ ,

$$[\mathbf{C}_m] = f_m [\mathbf{M}_m] + f_k [\mathbf{K}_m] \quad (3.136)$$

as defined previously in (3.125). Because of orthogonality with respect to  $[\mathbf{C}_m]$ , the uncoupled normal coordinate equations are

$$[\mathbf{K}'] \{\mathbf{U}'\} + [\mathbf{C}'] \left\{ \frac{d \mathbf{U}'}{dt} \right\} + [\mathbf{M}'] \left\{ \frac{d^2 \mathbf{U}'}{dt^2} \right\} = \{\mathbf{0}\} \quad (3.137)$$

The modal matrix  $[\mathbf{P}]$  is usually ‘mass normalised’, leading to

$$[\mathbf{M}'] = [\mathbf{I}]$$

$$[\mathbf{C}'] = (f_m + f_k \omega^2) [\mathbf{I}] \quad (3.138)$$

$$[\mathbf{K}'] = \omega^2 [\mathbf{I}]$$

By working in normal coordinates it has become necessary to take a constant  $\gamma$  for the complete mesh being analysed [see equation (3.126)], although  $\gamma$  could be varied from mode to mode. Since many real systems contain areas with markedly different damping properties, this is an undesirable feature of the method in practice (see Program 11.2).

### Inclusion of Forcing Terms

When  $\{\mathbf{f}(t)\}$  is non-zero, the right-hand side of a typical modal equation becomes

$$[\mathbf{K}'] \{\mathbf{U}'\} + [\mathbf{C}'] \left\{ \frac{d\mathbf{U}'}{dt} \right\} + [\mathbf{M}'] \left\{ \frac{d^2\mathbf{U}'}{dt^2} \right\} = [\mathbf{P}]^T \{\mathbf{F}(t)\} \quad (3.139)$$

For example, suppose that in a specific problem only degrees of freedom 10 and 12 are loaded with forces  $\cos \theta t$ . The  $j$ th row of (3.139) would be

$$\omega_j^2 U'_j + (f_m + f_k \omega_j^2) \frac{dU'_j}{dt} + \frac{d^2U'_j}{dt^2} = (P_{10,j} + P_{12,j}) \cos \theta t \quad (3.140)$$

or

$$\omega_j^2 U'_j + 2\gamma \omega_j \frac{dU'_j}{dt} + \frac{d^2U'_j}{dt^2} = P'_j \cos \theta t \quad (3.141)$$

The particular solution to this equation with stationary initial conditions is

$$U'_j = \frac{(\omega_j^2 - \theta^2) P'_j}{(\omega_j^2 - \theta^2)^2 + 4\gamma^2 \omega_j^2 \theta^2} \cos \theta t + \frac{2\gamma \omega_j \theta P'_j}{(\omega_j^2 - \theta^2)^2 + 4\gamma^2 \omega_j^2 \theta^2} \sin \theta t \quad (3.142)$$

The normal coordinates having been determined, the actual displacements can be recovered using (3.134).

For more general forcing functions (3.139) must be solved by other means, for example by one of the direct integration methods described next.

### 3.13.2 Newmark or Crank–Nicolson Method

If Rayleigh damping is assumed, a class of recurrence relations based on linear interpolation in time can again be constructed, involving the scalar parameter  $\theta$  which varies between 1/2 and 1 in the same way as was done for first-order problems.

If using an assembly technique, the equation (3.135) including a forcing term is written at both the ‘0’ and ‘1’ time stations,

$$\begin{aligned} [\mathbf{K}_m] \{\mathbf{U}\}_0 + (f_m [\mathbf{M}_m] + f_k [\mathbf{K}_m]) \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + [\mathbf{M}_m] \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 &= \{\mathbf{F}\}_0 \\ [\mathbf{K}_m] \{\mathbf{U}\}_1 + (f_m [\mathbf{M}_m] + f_k [\mathbf{K}_m]) \left\{ \frac{d\mathbf{U}}{dt} \right\}_1 + [\mathbf{M}_m] \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_1 &= \{\mathbf{F}\}_1 \end{aligned} \quad (3.143)$$

and assuming linear interpolation in time, giving

$$\begin{aligned}\{\mathbf{U}\}_1 &= \{\mathbf{U}\}_0 + \Delta t \left( (1 - \theta) \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + \theta \left\{ \frac{d\mathbf{U}}{dt} \right\}_1 \right) \\ \left\{ \frac{d\mathbf{U}}{dt} \right\}_1 &= \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + \Delta t \left( (1 - \theta) \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 + \theta \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_1 \right)\end{aligned}\quad (3.144)$$

Rearrangement of these equations and elimination of acceleration terms leads to the following three recurrence relations:

$$\begin{aligned}&\left[ \left( f_m + \frac{1}{\theta \Delta t} \right) [\mathbf{M}_m] + (f_k + \theta \Delta t) [\mathbf{K}_m] \right] \{\mathbf{U}\}_1 \\ &= \theta \Delta t \{\mathbf{F}\}_1 + (1 - \theta) \Delta t \{\mathbf{F}\}_0 + \left( f_m + \frac{1}{\theta \Delta t} \right) [\mathbf{M}_m] \{\mathbf{U}\}_0 \\ &+ \frac{1}{\theta} [\mathbf{M}_m] \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + (f_k - (1 - \theta) \Delta t) [\mathbf{K}_m] \{\mathbf{U}\}_0\end{aligned}\quad (3.145)$$

$$\left\{ \frac{d\mathbf{U}}{dt} \right\}_1 = \frac{1}{\theta \Delta t} (\{\mathbf{U}\}_1 - \{\mathbf{U}\}_0) - \frac{1 - \theta}{\theta} \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 \quad (3.146)$$

$$\left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_1 = \frac{1}{\theta \Delta t} \left( \left\{ \frac{d\mathbf{U}}{dt} \right\}_1 - \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 \right) - \frac{1 - \theta}{\theta} \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 \quad (3.147)$$

The algorithm requires initial conditions on displacements  $\{\mathbf{U}\}_0$  and velocities  $\{d\mathbf{U}/dt\}_0$  to be provided in order to get started.

In the special case when  $\theta = 1/2$  this method is Newmark's ' $\beta = 1/4$ ' method, which is also the exact equivalent of the Crank–Nicolson method used in first-order problems. There are other variants of the Newmark type, but this is the most common.

The principal recurrence relation (3.145) is clearly similar to those which arose in first-order problems, for example (3.100). Although substantially more matrix-by-vector multiplications are involved on the right-hand side, together with matrix and vector additions, the recurrence again consists essentially of an equation solution per time step. Advantage can as usual be taken of a constant left-hand-side matrix should this occur, and ‘element-by-element’ strategies are easily implemented via pcg (see Chapters 11 and 12).

### 3.13.3 Wilson's Method

Assuming again an assembly approach, the equations (3.143) are advanced from some known state  $\{\mathbf{U}\}_0$ ,  $\{d\mathbf{U}/dt\}_0$  and  $\{d^2\mathbf{U}/dt^2\}_0$  to the new solution  $\{\mathbf{U}\}_1$ ,  $\{d\mathbf{U}/dt\}_1$  and  $\{d^2\mathbf{U}/dt^2\}_1$  an interval  $\Delta t$  later by first linearly extrapolating to a hypothetical solution, say  $\{\mathbf{U}\}_2$ ,  $\{d\mathbf{U}/dt\}_2$  and  $\{d^2\mathbf{U}/dt^2\}_2$  an interval  $\delta t = \theta \Delta t$  later where  $1.4 \leq \theta \leq 2$ .

If Rayleigh damping is again assumed,  $\{\mathbf{U}\}_2$  is first computed from

$$\begin{aligned} & \left[ \left( \frac{6}{\theta^2 \Delta t^2} + \frac{3f_m}{\theta \Delta t} \right) [\mathbf{M}_m] + \left( \frac{3f_k}{\theta \Delta t} + 1 \right) [\mathbf{K}_m] \right] \{\mathbf{U}\}_2 = \{\mathbf{F}\}_2 \\ & + \left[ \left( \frac{6}{\theta^2 \Delta t^2} + \frac{3f_m}{\theta \Delta t} \right) \{\mathbf{U}\}_0 + \left( \frac{6}{\theta \Delta t} + 2f_m \right) \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + \left( 2 + \frac{f_m \theta \Delta t}{2} \right) \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 \right] [\mathbf{M}_m] \\ & + \left[ \frac{3f_k}{\theta \Delta t} \{\mathbf{U}\}_0 + 2f_k \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + \frac{f_k \theta \Delta t}{2} \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 \right] [\mathbf{K}_m] \end{aligned} \quad (3.148)$$

where

$$\{\mathbf{F}\}_2 = (1 - \theta)\{\mathbf{F}\}_0 + \theta\{\mathbf{F}\}_1 \quad (3.149)$$

The acceleration at the hypothetical station can then be computed from

$$\left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_2 = \frac{6}{\theta^2 \Delta t^2} (\{\mathbf{U}\}_2 - \{\mathbf{U}\}_0) - \frac{6}{\theta \Delta t} \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 - 2 \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 \quad (3.150)$$

and thus the acceleration at the true station can be interpolated or ‘averaged’ using

$$\left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_1 = \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 + \frac{1}{\theta} \left( \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_2 - \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 \right) \quad (3.151)$$

A Crank–Nicolson equation then gives the desired velocity from

$$\left\{ \frac{d\mathbf{U}}{dt} \right\}_1 = \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + \frac{\Delta t}{2} \left( \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 + \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_1 \right) \quad (3.152)$$

and the updated displacements from

$$\{\mathbf{U}\}_1 = \{\mathbf{U}\}_0 + \Delta t \left\{ \frac{d\mathbf{U}}{dt} \right\}_0 + \frac{\Delta t^2}{3} \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_0 + \frac{\Delta t^2}{6} \left\{ \frac{d^2\mathbf{U}}{dt^2} \right\}_1 \quad (3.153)$$

The principal recurrence relation (3.148) is again of the familiar type for all one-step time-integration methods.

### 3.13.4 Complex Response

The most troublesome feature of linearised solutions to (3.143) is the proper inclusion of damping. Observations indicate that damping can be frequency-independent and that it varies with material type and location, for example being much higher in soils than in steel or concrete and higher in areas of large (plastic) deformation than in elastic regions.

It has been shown that the modal superposition method is best suited to the analysis of systems where damping is uniformly distributed over the whole system for each mode. When damping is not uniformly distributed, the direct integration methods can be used with variable damping in each element (Idriss *et al.*, 1973) but it is still difficult to preserve frequency independence of the damping ratio  $\gamma$ . A method that does not suffer from either of these drawbacks is the complex response or complex transfer function method

(Timoshenko *et al.*, 1974) which is widely used in seismic analysis of soil–structure interaction (Lysmer *et al.*, 1974).

The method takes as its starting point the undamped equations of motion (3.123) and assumes that  $\{\mathbf{f}(t)\}$  is harmonic of the form

$$\{\mathbf{f}(t)\} = \{\mathbf{f}_a\} e^{i\omega t} \quad (3.154)$$

where  $\{\mathbf{f}_a\}$  is a (possibly complex) force amplitude vector. In a linearised system, this implies that the response  $\{\mathbf{u}\}$  is also harmonic

$$\{\mathbf{u}(t)\} = \{\mathbf{u}_a\} e^{i\omega t} \quad (3.155)$$

where  $\{\mathbf{u}_a\}$  is a (possibly complex) displacement amplitude vector.

When (3.155) are substituted into (3.123), the set of equations

$$[\mathbf{k}_m] - \omega^2 [\mathbf{m}_m] \{\mathbf{u}_a\} = \{\mathbf{f}_a\} \quad (3.156)$$

is obtained which can be solved for  $\{\mathbf{u}_a\}$  provided that  $[\mathbf{k}_m] - \omega^2 [\mathbf{m}_m]$  is non-singular or, in other words,  $\omega$  is not a natural frequency of the undamped system. The real and imaginary parts of the input  $\{\mathbf{f}_a\}$  and output  $\{\mathbf{u}_a\}$  correspond.

An advantage of the method is that damping can be introduced in the form of a complex equivalent of  $[\mathbf{k}_m]$  called  $[\mathbf{k}_m^*]$  such that

$$[\mathbf{k}_m^*] = [\mathbf{k}_m] \left(1 - 2\gamma^2 + 2i\gamma\sqrt{1 - \gamma^2}\right) \quad (3.157)$$

This produces the same amplitudes as a modal analysis with a damping ratio of  $\gamma$  and to a close approximation the same phase. Clearly the element-level equations

$$[\mathbf{k}_m^*] - \omega^2 [\mathbf{m}_m] \{\mathbf{u}_a\} = \{\mathbf{f}_a\} \quad (3.158)$$

with a simple harmonic input with a single frequency  $\omega$ , can be assembled in the usual way using varying  $\gamma$  values when required. Assembly leads to a global system of complex simultaneous equations requiring specialised assembly and solution routines to account for the real and imaginary parts (see Program 11.5).

Solution of the equations leads to the displacement amplitudes from which the displacement time histories can be obtained from the global counterpart of (3.155).

More usually the loading function  $\{\mathbf{f}\}$  is a general non-harmonic function, for example a seismic record or a series of ocean wave height measurements. The complex response method can still be used if  $\{\mathbf{f}\}$  is approximated by a series of equivalent harmonic components.

### 3.13.5 Explicit Methods and Other Storage-Saving Strategies

The implicit methods described above are relatively safe to use due to their unconditional stability. However, as was the case for first-order problems, storage demands become considerable for large systems, and so can solution times for non-linear problems [even though refactorisation of the left-hand side of (3.145) or (3.148) is not usually necessary, the non-linear effects having been transposed to the right-hand side].

Using a pcg strategy, the implicit equation solution can always be done ‘element-by-element’, but the simplest option is the analogue of (3.104), in which  $\theta$  is set to zero

and the mass matrix lumped. In the resulting explicit algorithm, operations are carried out element-wise and no global system storage is necessary (see, e.g., Program 11.8). Of course the drawback is potential loss of stability, so that stable time steps may need to be very small indeed.

Since stability is governed by the highest natural frequency of the numerical approximation and since such high frequencies are derived from the stiffest elements in the system, it is quite possible to implement hybrid methods in which the very stiff elements are integrated implicitly, but the remainder are integrated explicitly. Equation solution as implied by (3.145), for example, is still necessary, but the half-bandwidth of the rows in the coefficient matrix associated with freedoms in explicit elements not connected to implicit ones is only one. Thus great savings in storage can be made (Smith, 1984).

Another alternative is to resort to operator splitting, as was done in first-order problems. In Chapter 11, implicit, explicit and mixed implicit/explicit algorithms are described and listed, with alternative assembly or EBE solution for the implicit cases. Although product EBE methods have been developed (Wong *et al.*, 1989), they are beyond the scope of the present book.

## References

- Arnoldi WE 1951 The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Q Appl Math* **9**, 17–29.
- Bai Z, Demmel J, Dongarra J, Ruhe A and der Vorst HV 2000 *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM Press, Philadelphia.
- Bathe KJ and Wilson EL 1996 *Numerical Methods in Finite Element Analysis*, 3rd edn. Prentice-Hall, Englewood Cliffs, NJ.
- Cardoso JP 1994 *Generation of finite element matrices using computer algebra*. Master's thesis, School of Engineering, University of Manchester.
- Chan SH, Phoon KK and Lee FH 2001 A modified Jacobi preconditioner for solving ill-conditioned Biot's consolidation equations using symmetric quasi-minimal residual method. *Int J Numer Anal Methods Geomech* **25**(10), 1001–1025.
- Ergatoudis J, Irons BM and Zienkiewicz OC 1968 Curved isoparametric quadrilateral elements for finite element analysis. *Int J Solids Struct* **4**, 31.
- Gladwell I, Smith IM, Gilvary B and Wong SW 1989 A consistent mass EBE algorithm for linear parabolic systems. *Comm Appl Numer Methods* **5**, 229–235.
- Greenbaum A 1997 *Iterative Methods for Solving Linear Systems*. SIAM Press, Philadelphia.
- Griffiths DV 1991 Generalised numerical integration of moments. *Int J Numer Methods Eng* **32**(1), 129–147.
- Griffiths DV 1994a Coupled analyses in geomechanics, In *Visco-plastic Behavior of Geomaterials* (eds Cristescu ND and Gioda G). Springer-Verlag Wien, New York, pp. 245–317.
- Griffiths DV 1994b Stiffness matrix of the four-node quadrilateral element in closed-form. *Int J Numer Methods Eng* **37**(6), 1027–1038.
- Griffiths DV 2004 Use of computer algebra systems in finite element software development. In *Proc 7th Int Congress on Numerical Methods in Engineering and Scientific Applications, CIMENICS'04* (eds Rojo J *et al.*), Sociedad Venezolana de Métodos Numéricos en Ingeniería Caracas, Venezuela, pp. CI 55–66.
- Griffiths DV and Smith IM 2006 *Numerical Methods for Engineers*, 2nd edn. Chapman & Hall/CRC Press, Boca Raton, FL.
- Griffiths DV, Huang J and Schierneyer RP 2009 Elastic stiffness of straight-sided triangular finite elements by analytical and numerical integration. *Comm Num Meth Eng* **25**(3), 247–262.
- Hicks MA 1995 MONICA—a computer algorithm for solving boundary value problems using the double hardening constitutive model Monot: I algorithm development. *Int J Numer Anal Methods Geomech* **19**, 11–27.
- HSL 2011 A collection of Fortran codes for large scale scientific computation. <http://www.hsl.rl.ac.uk>.
- Hughes TJR, Levit I and Winget J 1983 Element by element implicit algorithms for heat conduction. *J Eng Mech, ASCE* **109**(2), 576–585.

- Idriss IM, Lysmer J, Hwang R and Seed HB 1973 QUAD-4: A computer program for evaluating the seismic response of soil structures by variable damping finite element procedures. Technical Report EERC 73-16, University of California, Berkeley.
- Irons BM 1966 Engineering applications of numerical integration in stiffness method. *J Am Inst Aeronaut Astronaut* **14**, 2035.
- Irons BM 1971a *Numerical integration applied to finite element methods*. Department of Civil Engineering, University College of Swansea.
- Irons BM 1971b Quadrature rules for brick-based finite elements. *Int J Numer Methods Eng* **3**, 293–294.
- Jennings A and McKeown JJ 1992 *Matrix Computation*, 2nd edn. John Wiley & Sons, Chichester.
- Kelley CT 1995 *Iterative Methods for Linear and Nonlinear Equations*. SIAM Press, Philadelphia.
- Kidger DJ and Smith IM 1992 Eigenvalues and eigenmodes of 8-node brick elements. *Comm App Num Meth* **8**, 193–205.
- Kopal A 1961 *Numerical Analysis*, 2nd edn. Chapman & Hall, London.
- Lehoucq RB, Sorensen DC and Yang C 1998 *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM Press, Philadelphia.
- Lysmer J, Uda T, Seed H and Hwang R 1974 LUSH: A computer program for complex response analysis of soil–structure systems. Technical Report EERC 74-4, University of California, Berkeley.
- Parlett BN and Reid JK 1981 Tracking the progress of the Lanczos algorithm for large symmetric eigenproblems. *IMA J Numer Anal* **1**, 135–155.
- Sandhu RS and Wilson EL 1969 Finite-element analysis of seepage in elastic media. *J Eng Mech, ASCE* **95**(EM3), 641–652.
- Sleijpen GLG, van der Vorst HA and Fokkema DR 1994 BiCGStab(l) and other hybrid Bi-CG methods. *Numer Alg* **7**, 75–109.
- Smith IM 1979 Discrete element analysis of pile instability. *Int J Numer Anal Methods Geomech* **3**, 205–211.
- Smith IM 1984 Adaptability of truly modular software. *Eng Comput* **1**(1), 25–35.
- Smith IM 2000 A general purpose system for finite element analyses in parallel. *Eng Comput* **17**(1), 75–91.
- Smith IM and Kidger DJ 1991 Properties of the 20-node brick element. *Int J Numer Anal Methods Geomech* **15**(12), 871–891.
- Smith IM and Kidger DJ 1992 Elastoplastic analysis using the 14-node brick element family. *Int J Numer Methods Eng* **35**, 1263–1275.
- Smith IM, Wong SW, Gladwell I and Gilvary B 1989 PCG methods in transient FE analysis Part I: First order problems. *Int J Numer Methods Eng* **28**(7), 1557–1566.
- Taig IC 1961 Structural analysis by the matrix displacement method. Technical Report SO17, English Electric Aviation Report, Preston.
- Timoshenko SP, Young D and Weaver W 1974 *Vibration Problems in Engineering*, 4th edn. John Wiley & Sons, Chichester.
- Wong SW, Smith IM and Gladwell I 1989 PCG methods in transient FE analysis Part II: Second order problems. *Int J Numer Methods Eng* **28**(7), 1567–1576.
- Zienkiewicz OC, Irons BM, Ergatoudis J, Ahmad S and Scott FC 1969 Isoparametric and associated element families for two and three dimensional analysis. In *Proceedings of a Course on Finite Element Methods in Stress Analysis* (eds Holland I and Bell K). Norwegian University of Science and Technology, Trondheim.
- Zienkiewicz OC, Too J and Taylor RL 1971 Reduced integration technique in general analysis of plates and shells. *Int J Numer Methods Eng* **3**, 275–290.

# 4

# Static Equilibrium of Structures

## 4.1 Introduction

Practical finite element analysis had as its starting point matrix analysis of ‘structures’, by which engineers usually mean assemblages of elastic, line elements. The matrix displacement (stiffness) method is a special case of finite element analysis and, since many engineers still begin their acquaintance with the finite element method in this way, the opening applications chapter of this book is devoted to ‘structural’ analysis.

The first program, Program 4.1, permits the analysis of a rod subjected to combinations of axial loads and displacements at various points along its length. Each 1D rod element can have a different length and axial stiffness but the element stiffness matrices, being simple functions of these two quantities, are easily formed by a subroutine. Indeed, in nearly all the programs in this chapter, the element stiffness matrices consist of simple explicit expressions which are conveniently provided by subroutines. Program 4.2 introduces a more general treatment of rod elements, allowing analyses to be performed of 2D or 3D pin-jointed frames.

Program 4.3 permits the analysis of slender beams subjected to combinations of transverse and moment loading. Optionally, the program allows the inclusion of an elastic foundation enabling analysis of problems generally classified as ‘beams on elastic foundations’.

When 1D beam and rod elements are superposed, the result is a ‘beam–rod’ element, which is a powerful general element that can sustain axial, transverse and moment loading. This element is able to analyse all conventional structural frames. Program 4.4 implements the ‘beam–rod’ elements in the analysis of 2D or 3D framed structures.

Program 4.5 introduces material non-linearity in the form of an elastic–perfectly plastic moment/curvature relationship for beams. The program can compute plastic collapse of 1D, 2D or 3D structures when subjected to incrementally changing loads. The non-linearity is dealt with using an iterative, constant stiffness (modified Newton–Raphson) approach. Unlike more traditional approaches, at each iteration the internal loads on the structure are altered rather than the stiffness matrix itself. This approach will be used extensively in the elastic–plastic analyses of solids described in later chapters of the book, notably Chapter 6.

Program 4.6 performs elastic stability analysis of axially loaded beams. As in Program 4.3, an elastic foundation is optional. The program computes the lowest

buckling load by iteratively obtaining the smallest eigenvalue of the system based on its stiffness and geometric properties.

The final program in the chapter, Program 4.7, describes a method for analysing rectangular thin plates in bending. This could be considered to be the first ‘genuine’ finite element in the book. The plate stiffness matrix is formed using numerical integration, anticipating the solid mechanics applications of Chapter 5 and beyond.

In the interests of generality and portability, all programs in the book assume that the data and results have the file types `*.dat` and `*.res`, which are assigned to channels 10 and 11, respectively. In all programs described in this book, the input data file name is read from the command line by subroutine `getname` (see Appendix D) and assigned to all output files. For example, if the user creates a data file called `fred.dat`, the results file will be held in `fred.res`. Some programs in later chapters produce simple graphical files in Postscript with extensions such as `*.msh`, `*.dis`, `*.vec` and `*.con`. In these cases, the data file name is again used, so the Postscript files will have names like `fred.msh`, etc.

Further explanation of files provided online to streamline the above process can be found at the software home page described in the Preface.

## Program 4.1 One-dimensional analysis of axially loaded elastic rods using 2-node rod elements

```
PROGRAM p41
!-----
! Program 4.1 One dimensional analysis of axially loaded elastic rods
!           using 2-node rod elements.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndof=2,nels,neq,nlen,nod=2, &
nodof=1,nn,nprops=1,np_types,nr
REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:)),kdiag(:,nf(:,:,no(:), &
node(:),num(:)
REAL(iwp),ALLOCATABLE::action(:,eld(:,ell(:,km(:,:,kv(:,loads(:, &
prop(:, :,value(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,np_types; nn=nels+1
ALLOCATE(g(ndof),num(nod),nf(nodof,nn),etype(nels),ell(nels),eld(ndof), &
km(ndof,ndof),action(ndof),g_g(ndof,nel),prop(nprops,np_types))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')
&
" There are",neq," equations and the skyline storage is",kdiag(neq)
```

```

!-----global stiffness matrix assembly-----
kv=zero
elements_2: DO iel=1,nels
    CALL rod_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
    CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read loads and/or displacements-----
loads=zeros; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
READ(10,*)fixed Freedoms
IF(fixed Freedoms/=0)THEN
    ALLOCATE(node(fixed Freedoms),no(fixed Freedoms),value(fixed Freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed Freedoms)
    DO i=1,fixed Freedoms; no(i)=nf(1,node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution -----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
WRITE(11,'(/A)')" Node Disp"
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(nf(:,k)); END DO
!-----retrieve element end actions-----
WRITE(11,'(/A)')" Element Actions"
elements_3: DO iel=1,nels
    CALL rod_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
    eld=loads(g); action=MATMUL(km,eld); WRITE(11,'(I5,2E12.4)')iel,action
END DO elements_3
STOP
END PROGRAM p41

```

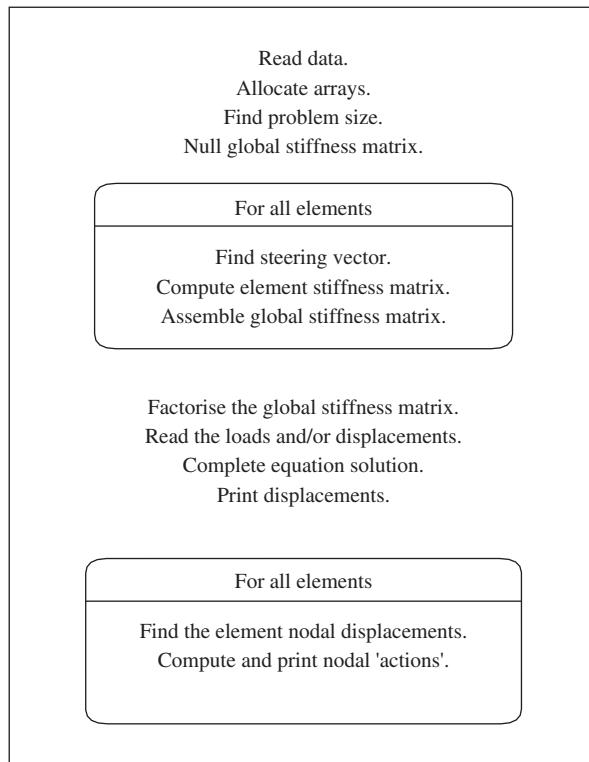
The main features of this program are the elastic rod element stiffness matrix (2.11) and the global stiffness matrix assembly described in Section 3.7. The structure chart in Figure 4.1 gives the main sequence of operations.

Program 4.1 is illustrated by two examples shown in Figures 4.2 and 4.3. In both cases, the rod is restrained at one end and free at the other. The rod in Figure 4.2 is subjected to a uniformly distributed axial load (units of force per unit length), and the rod in Figure 4.3 is subjected to a fixed displacement at its tip.

Each node has one degree of freedom, namely the axial displacement. The global node and element numbering systems read from the left and, at the element level, node one is always to the left and node two to the right as shown in Figure 4.4. As explained in Chapter 3, the nodal freedom numbering associated with each element, accounting for any restraints, is contained in the ‘steering’ vector  $g$ . Thus, with reference to Figures 4.3 and 4.4,  $g$  for element one would be  $[0 \ 1]^T$  and for element two  $[1 \ 2]^T$ , and so on.

If a zero appears in the ‘steering’ vector  $g$ , this means that the corresponding displacement is fully fixed and equal to zero, as at the right end of the example in Figure 4.2, and the left end of the example in Figure 4.3. In such cases, the ‘zero freedom’ is not assembled into the global matrices.

Nodal freedom data concerning boundary restraints is read by the main program, and takes the form of the number of restrained nodes,  $nr$ , followed by, for each restrained node, the restrained node number and a zero. The default is that nodes are not restrained. For problems such as this where 1D elements are strung together in a line, it is a simple matter to automate the generation of the  $g$  vector for each element. This is done by the library subroutine `num_to_g` which picks the correct entries out of the nodal freedom array  $nf$  generated by subroutine `formnf` (See Appendices D and E for description of library subroutines).



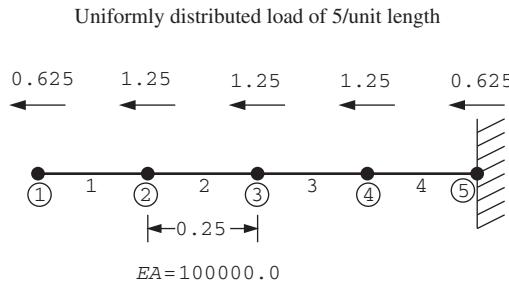
**Figure 4.1** Structure chart for Program 4.1

Many programs in the book use this approach for defining boundary conditions. In cases later on, where nodes have more than one degree of freedom, some of those nodal freedoms may be restrained and others not. In these cases, the convention will then be to give the node number followed by either ones or zeros (in the correct sequence), where the latter implies a restrained (zero) degree of freedom and the former an unrestrained freedom.

Returning to the main program, some scalar quantities are defined by the type of element being used. For example, there can only be two nodes per element, so `nod=2` and these are assigned in the declaration lines.

The ‘input and initialisation’ section reads the number of elements `nels` and the number of property types `np_types`. If there is only one property type (`np_types=1`) as in the example shown in Figure 4.2, then the property is read and automatically allocated to all elements. If there is more than one property type, as in the example shown in Figure 4.3, then the properties are read, followed by the reading of an integer vector `etype` which holds information on which property is assigned to which element. The length of each element is read into the real vector `e11`, and the boundary condition data is read enabling the formation of array `nf`.

Inside the ‘global stiffness matrix assembly’ section, the element stiffnesses and lengths are used by subroutine `rod_km` to compute the element stiffness matrices `km`.



```

nels      np_types
4          1

prop (ea)
100000.0

etype(not needed)

ell
0.25  0.25  0.25  0.25

nr, (k, nf(:,k), i=1, nr)
1
5 0

loaded_nodes, (k, loads(nf(:,k)), i=1, loaded_nodes)
5
1 -0.625  2 -1.25  3 -1.25  4 -1.25  5 -0.625

fixed_freedoms
0

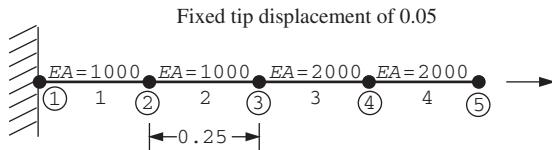
```

**Figure 4.2** Mesh and data for first Program 4.1 example

The global stiffness matrix (stored as a skyline column vector  $\mathbf{kv}$ , see Section 3.7.10) is then assembled for all elements in turn by the library subroutine `fspdrv` once the ‘steering’ vector  $\mathbf{g}$  has been retrieved. Gaussian elimination on the system equations is split into a ‘factorisation’ phase performed by library subroutine `sparin` and a forward and back-substitution phase performed by library subroutine `spabac`. The ‘loading’ data is then read, and this takes the form of information relating to nodal forces and/or fixed nodal displacements.

In the case of loads, `loaded_nodes` is read first signifying the number of nodes with forces applied. Then for each of these, the node number and the applied force are read. In this rod example there is only one freedom at each node, but in later programs in the chapter where more than one freedom exists at each node, all the ‘forces’ applied at the loaded node must be included in the correct sense (even if some of them are zero).

In the case of fixed displacements, `fixed_freedoms` is read, signifying the number of fixed freedoms in the mesh. Then for each of these, the node number and the value to which the freedom is to be fixed is read. In later programs in the chapter where more than one freedom exists at each node, data must also be read into the vector `sense`, which gives the sequential number of the freedom at the node that is to be fixed. If a particular node has more than one fixed freedom, the node must be entered in the data list for each fixed ‘sense’. If either `loaded_nodes` or `fixed_freedoms` equals zero, no further data relating to that category is required.



```

nels      np_types
4          2

prop (ea)
2000.0  1000.0

etype
2 2 1 1

ell
0.25  0.25  0.25  0.25

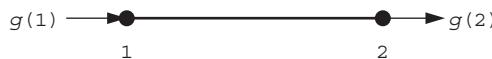
nr,(k,nf(:,k),i=1,nr)
1
1 0

loaded_nodes
0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
1
5  0.05

```

**Figure 4.3** Mesh and data for second Program 4.1 example



**Figure 4.4** Node and freedom numbering for rod elements

In the case shown in Figure 4.2, a rod of uniform stiffness equal to  $10^5$  is subjected to a (negative) uniformly distributed axial load of 5 (units of force per unit length). The force has been ‘lumped’ at the nodes as was indicated in (2.10). In this example, there are no fixed displacements, so `fixed_freedoms` is read as zero.

In the case shown in Figure 4.3, the rod has a non-uniform stiffness, and since there is more than one property group (`np_types = 2`), the element type vector `etype` must be read indicating in this case that elements 1 and 2 have an axial stiffness of 1000.0, and elements 3 and 4 have an axial stiffness of 2000.0. There are no loaded nodes so `loaded_nodes` is read as zero but a fixed displacement of 0.05 is applied to the tip of the rod, so `fixed_freedoms` is read as 1, followed by the node number (5) and the magnitude of the fixed displacement (0.05).

Following equation solution, the nodal displacements (overwritten as `loads`) are computed and printed. A final ‘post-processing’ phase is then performed, in which the elements are scanned once more. In this loop, the element nodal displacements (`e1d`) are retrieved from the global displacements vector and the element stiffness matrices (`km`) re-computed. Multiplication of the element nodal displacements by the element stiffness matrix using `MATMUL` results in the element ‘actions’ vector called `action`, which holds the internal end reaction forces for each element.

(a) There are 4 equations and the skyline storage is 7

Node	Disp
1	-0.2500E-04
2	-0.2344E-04
3	-0.1875E-04
4	-0.1094E-04
5	0.0000E+00

Element	Actions
1	-0.6250E+00 0.6250E+00
2	-0.1875E+01 0.1875E+01
3	-0.3125E+01 0.3125E+01
4	-0.4375E+01 0.4375E+01

(b) There are 4 equations and the skyline storage is 7

Node	Disp
1	0.0000E+00
2	0.1667E-01
3	0.3333E-01
4	0.4167E-01
5	0.5000E-01

Element	Actions
1	-0.6667E+02 0.6667E+02
2	-0.6667E+02 0.6667E+02
3	-0.6667E+02 0.6667E+02
4	-0.6667E+02 0.6667E+02

**Figure 4.5** Results from (a) first and (b) second Program 4.1 examples

The computed results for both cases are reproduced in Figure 4.5. In the first case, the end deflection at node 1 is given as  $-0.25 \times 10^{-4}$  which is the exact solution. To retrieve the correct internal forces, the equivalent nodal loads for each element must be subtracted from the ‘actions’ printed in the results file. For element 1, for example, this results in a tip force at node 1 of zero, and an internal tensile force at node 2 of 1.25.

In the second case, the nodal displacements indicate the distribution of displacements along the length of the rod up to the end node 5, which has the expected displacement of 0.05. All the elements in the rod are sustaining a tensile load of 66.67.

In problems such as this, the nodal displacements are always in exact agreement with the closed-form solution achieved by direct integration of the governing equation. Between the nodes, however, the solution may be approximate, due to the limitations of the shape functions (linear in this case).

## Program 4.2 Analysis of elastic pin-jointed frames using 2-node rod elements in two or three dimensions

```
PROGRAM p42
!-----
! Program 4.2 Analysis of elastic pin-jointed frames using 2-node rod
!           elements in 2- or 3-dimensions
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER, PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim,ndof=2,nels,neq,nlen,  &
nod=2,nodof,nn,nprops=1,np_types,nr
REAL(iwp)::axial,penalty=1.0e20_iwp,zero=0.0_iwp; CHARACTER(LEN=15)::argv
```

```

!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:,g_g(:,:,),g_num(:,:,),kdiag(:),nf(:,:,), &
 no(:),node(:),num(:),sense(:)
REAL(iwp),ALLOCATABLE::action(:),coord(:,:),eld(:),g_coord(:,:,),km(:,:,), &
 kv(:,loads(:,prop(:,:,),value(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,nn,ndim,np_types; nodof=ndim; ndof=nod*nodof
ALLOCATE(nf(nodof,nn),km(ndof,ndof),coord(nod,ndim),g_coord(ndim,nn), &
 eld(ndof),action(ndof),g_num(nod,nels),num(nod),g(ndof),g_g(ndof,nels),&
 etype(nels),prop(nprops,np_types))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)g_coord; READ(10,*)g_num
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')                                     &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global stiffness matrix assembly-----
kv=zero
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    CALL pin_jointed(km,prop(1,etype(iel)),coord); g=g_g(:,iel)
    CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read loads and/or displacements-----
loads=zeros; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),no(fixed_freedoms), &
        sense(fixed_freedoms),value(fixed_freedoms))
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution -----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
WRITE(11,'(/A)')    " Node Displacement(s)"
DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----retrieve element end actions-----
WRITE(11,'(/A)')" Element Actions"
elements_3: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    CALL pin_jointed(km,prop(1,etype(iel)),coord); g=g_g(:,iel)
    eld=loads(g); action=MATMUL(km,eld); WRITE(11,'(I5,6E12.4)')iel,action
    CALL glob_to_axial(axial,action,coord)
    WRITE(11,'(A,E12.4)')" Axial force =",axial
END DO elements_3
STOP
END PROGRAM p42

```

This program is an adaptation of the previous one to allow analysis of rod elements in two or three dimensions. Since rod elements can only sustain axial loads, the types of structural systems for which this is applicable are pin-jointed frames (in 2D) or space structures (in 3D).

The previous program considered rod elements in 1D joined end to end, in which the number of nodes was always one greater than the number of elements. This will no longer be true for general 2D or 3D structures, so the number of nodes `nn` is now included as data together with the dimensionality of the problem `ndim`.

The variable names are virtually the same as in the previous program. The real array `e11` has been discarded because the rod element lengths are more conveniently calculated from their nodal coordinates. The variable `axial` has been introduced to hold the axial force sustained by each member, and the integer vector `sense` holds the sense of any nodal fixed freedoms, since there is now more than one freedom at each node.

Nodal coordinates must be provided as data and read directly into array `g_coord`, followed by data relating to the element node numbers, read into `g_num`. Two new library subroutines are also introduced. Subroutine `pin_jointed` computes the element stiffness matrix `km`, and subroutine `glob_to_loc` transforms the element ‘actions’ into the axial force held in `axial`.

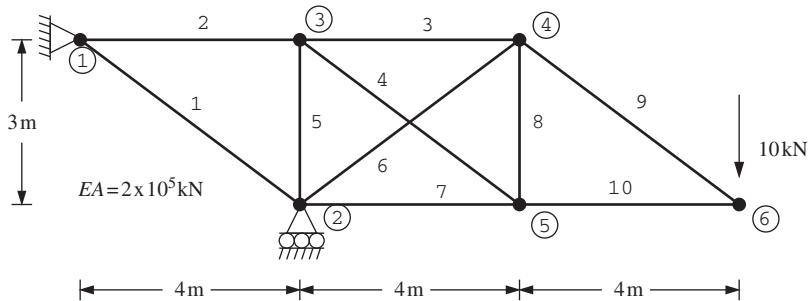
The first example to be solved by Program 4.2 is the 2D pin-jointed frame shown in Figure 4.6. Each element now has four degrees of freedom with an *x*- and a *y*-translation permitted at each node as shown in Figure 4.7.

In a small problem such as this, the order in which the nodes are numbered is immaterial. In larger problems, however, nodal numbering should be made in the most economical order in order to minimise the skyline bandwidths and hence the storage requirements (see Section 3.7.10). In very large problems which use an assembly strategy, a bandwidth optimiser would be used. The loaded nodes and fixed freedoms data follow the same procedure as described in the previous program. In this example, a single load of  $-10.0$  is applied in the *y*-direction at node 6. Note how the load in the *x*-direction at this node has to be read in as zero. There are no fixed freedoms in this example, hence `fixed_freedoms` is read as zero, indicating no further data is needed.

The results given in Figure 4.8 indicate the nodal displacements, followed by the end ‘actions’ and axial force in each element. The results indicate that the displacement under the load is  $-0.007263$  and the axial load in element number 1 is  $-33.33$  (compressive).

The second example to be solved by Program 4.2 is the 3D pin-jointed frame shown in Figure 4.9. Each element now has six degrees of freedom with an *x*-, *y*-and *z*-translation permitted at each node as shown in Figure 4.10.

The data organisation is virtually the same as in the previous example. The dimensionality is increased to `ndim=3` and there are correspondingly three coordinates at each node and three freedoms to be defined at each restrained node. The space-frame shown in Figure 4.9 represents a pyramid-like structure loaded by a force at its apex with components in the *x*-, *y*- and *z*-directions of  $20$ ,  $-20$  and  $30$ , respectively. In addition, the *y*-freedom (sense 2) at the apex is displaced by  $-0.0005$ . The computed results shown in Figure 4.11 indicate that the corresponding displacement components of the loaded node are  $0.2569 \times 10^{-3}$ ,  $-0.5 \times 10^{-3}$  (as would be expected) and  $0.7614 \times 10^{-4}$ . The axial force in element number 2 is computed to be  $49.57$  (tensile). It may be noted that in cases such as this where the data provides a load *and* a fixed displacement at a particular freedom, the value of the load is immaterial, and the displacement always takes precedence.



```

nels   nn    ndim   np_types
10      6      2       1

prop(ea)
2.0e5

etype(not needed)

g_coord
0.0 3.0    4.0 0.0    4.0 3.0
8.0 3.0    8.0 0.0   12.0 0.0

g_num
1 2   1 3   3 4   3 5   3 2
2 4   2 5   5 4   4 6   5 6

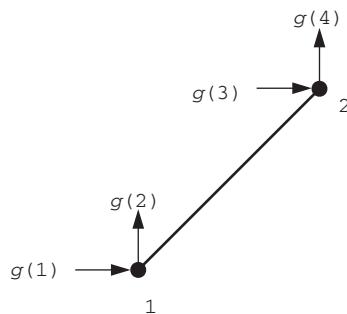
nr, (k,nf(:,k),i=1,nr)
2
1 0 0   2 1 0

loaded_nodes, (k,loads(nf(:,k)))
1
6   0.0 -10.0

fixed_freedoms
0

```

**Figure 4.6** Mesh and data for first Program 4.2 example



**Figure 4.7** Node and freedom numbering for 2D rod elements

There are 9 equations and the skyline storage is 39

Node	Displacement (s)		
1	0.0000E+00	0.0000E+00	
2	-0.1042E-02	0.0000E+00	
3	0.5333E-03	-0.6562E-04	
4	0.9500E-03	-0.3046E-02	
5	-0.1425E-02	-0.2981E-02	
6	-0.1692E-02	-0.7263E-02	

Element	Actions		
1	0.2667E+02	-0.2000E+02	-0.2667E+02
	Axial force =	-0.3333E+02	
2	-0.2667E+02	0.0000E+00	0.2667E+02
	Axial force =	0.2667E+02	
3	-0.2083E+02	0.0000E+00	0.2083E+02
	Axial force =	0.2083E+02	
4	-0.5833E+01	0.4375E+01	0.5833E+01
	Axial force =	0.7292E+01	-0.4375E+01
5	0.0000E+00	-0.4375E+01	0.0000E+00
	Axial force =	-0.4375E+01	0.4375E+01
6	0.7500E+01	0.5625E+01	-0.7500E+01
	Axial force =	-0.9375E+01	-0.5625E+01
7	0.1917E+02	0.0000E+00	-0.1917E+02
	Axial force =	-0.1917E+02	0.0000E+00
8	0.0000E+00	0.4375E+01	0.0000E+00
	Axial force =	-0.4375E+01	-0.4375E+01
9	-0.1333E+02	0.1000E+02	0.1333E+02
	Axial force =	0.1667E+02	-0.1000E+02
10	0.1333E+02	0.0000E+00	-0.1333E+02
	Axial force =	-0.1333E+02	0.0000E+00

Figure 4.8 Results from first Program 4.2 example

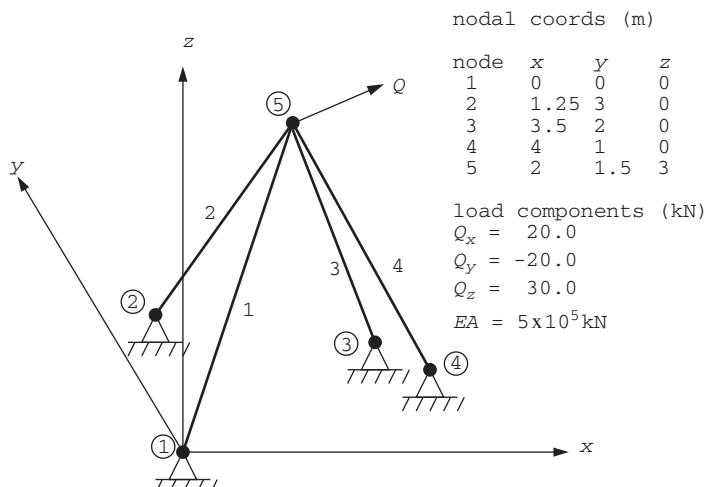


Figure 4.9 Mesh and data for second Program 4.2 example (Continued)

```

nels nn ndim np_types
4 5 3 1

prop(ea)
5.0e5

etype(not needed)

g_coord
0.0 0.0 0.0 1.25 3.0 0.0 3.5 2.0 0.0
4.0 1.0 0.0 2.0 1.5 3.0

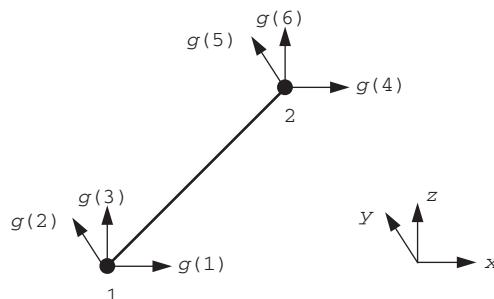
g_num
1 5 2 5 3 5 4 5

nr,(k,nf(:,k),i=1,nr)
4
1 0 0 0 2 0 0 0 3 0 0 0 4 0 0 0

loaded_nodes,(k,loads(nf(:,k)))
1
5 20.0 -20.0 30.0

fixed Freedoms,(node(i),sense(i),value(i),i=1,fixed Freedoms)
1
5 2 -0.0005

```

**Figure 4.9 (Continued)****Figure 4.10** Node and freedom numbering for 3D rod elements

There are 3 equations and the skyline storage is 6

Node	Displacement(s)		
1	0.0000E+00	0.0000E+00	0.0000E+00
2	0.0000E+00	0.0000E+00	0.0000E+00
3	0.0000E+00	0.0000E+00	0.0000E+00
4	0.0000E+00	0.0000E+00	0.0000E+00
5	0.2569E-03	-0.5000E-03	0.7614E-04

Element Actions
1 0.1298E+00 0.9735E-01 0.1947E+00 -0.1298E+00 -0.9735E-01 -0.1947E+00 Axial force = -0.2534E+00
2 -0.1082E+02 0.2163E+02 -0.4327E+02 0.1082E+02 -0.2163E+02 0.4327E+02 Axial force = 0.4957E+02
3 0.1789E+01 0.5963E+00 -0.3578E+01 -0.1789E+01 -0.5963E+00 0.3578E+01 Axial force = 0.4045E+01
4 -0.1110E+02 0.2775E+01 0.1665E+02 0.1110E+02 -0.2775E+01 -0.1665E+02 Axial force = -0.2021E+02

**Figure 4.11** Results from second Program 4.2 example

### Program 4.3 Analysis of elastic beams using 2-node beam elements (elastic foundation optional)

```

PROGRAM p43
!-----
! Program 4.3 Analysis of elastic beams using 2-node beam elements
! (elastic foundation optional).
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndof=4,nels,neq,nlen,nod=2, &
nodof=2,nn,nprops,np_types,nr
REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp;; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:),g_g(:,:,),kdiag(:),nf(:,:,),no(:), &
node(:),num(:),sense(:)
REAL(iwp),ALLOCATABLE::action(:,eld(:,ell(:,km(:,:,),kv(:,loads(:, &
mm(:,:,prop(:,:,value(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,nprops,np_types; nn=nels+1
ALLOCATE(g(ndof),num(nod),nf(nodof,nn),etype(nels),ell(nels),eld(ndof), &
km(ndof,ndof),mm(ndof,ndof),action(ndof),g_g(ndof,nels), &
prop(nprops,np_types))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))') &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global stiffness matrix assembly-----
kv=zero
elements_2: DO iel=1, nels
    CALL beam_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
    mm=zero; IF(nprops>1)CALL beam_mm(mm,prop(2,etype(iel)),ell(iel))
    CALL fsparv(kv,km+mm,g,kdiag)
END DO elements_2
!-----read loads and/or displacements-----
loads=zeros; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),no(fixed_freedoms), &
    sense(fixed_freedoms),value(fixed_freedoms))
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution -----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
WRITE(11,'(/A)')" Node Translation Rotation"
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(nf(:,k)); END DO

```

```

!-----retrieve element end actions-----
WRITE(11,'(/A)')" Element Force      Moment      Force      Moment"
elements_3: DO iel=1,ne1s
  CALL beam_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
  mm=zero; IF(nprops>1)CALL beam_mm(mm,prop(2,etype(iel)),ell(iel))
  eld=loads(g); action=MATMUL(km+mm,eld)
  WRITE(11,'(I5,4E12.4)')iel,action
END DO elements_3
STOP
END PROGRAM p43

```

This program has much in common with Program 4.1 for rod elements. A line of beam elements of different stiffnesses and lengths, optionally resting on an elastic foundation, can be analysed for any combination of transverse and/or moment loading.

The inclusion of an elastic foundation is signalled by reading `nprops` as 2, indicating that two properties, namely the beam flexural stiffness  $EI$  and the foundation stiffness  $k$ , must be read into the properties matrix `prop`. If `nprops` is read as 1, however, only one property, the flexural stiffness  $EI$ , is required as data and the analysis is of a simple beam.

The beam element stiffness matrix (Section 2.4.1) is provided by subroutine `beam_km`, and the foundation stiffness matrix, closely related to the element ‘mass’ matrix (Section 2.4.2), by subroutine `beam_mm`. The real array `mm` is included in the program to hold the foundation stiffness matrix. As in Program 4.1, the length of each element is read into the real vector `ell`.

The first example problem shown in Figure 4.12 represents a non-uniform beam subjected to a combination of nodal loads and fixed displacements. Node 1 is to be rotated clockwise by 0.001 and node 3 is to be translated vertically downwards by 0.005. In addition, a vertical force of 20 acts at node 2, a uniformly distributed load of 4/unit length acts between nodes 3 and 4, and a linearly decreasing load of 4/unit length to zero acts between nodes 4 and 5.

At each node, two degrees of freedom are possible, a vertical translation and a rotation, in that order. The global node and element numbering reads from left to right, and at the element level, node one is always to the left and node two to the right as shown in Figure 4.13. Each element has four degrees of freedom taken in the order  $w_1$ ,  $\theta_1$ ,  $w_2$  and  $\theta_2$ . The nodal freedom numbering associated with each element, accounting for any restraints, is as usual contained in the ‘steering’ vector `g`. Thus, with reference to Figures 4.12 and 4.13, the steering vector for element 1 would be  $[0\ 1\ 2\ 3]^T$  and for element 2 would be  $[2\ 3\ 4\ 5]^T$ , and so on. It should be noted that in the ‘penalty’ or ‘stiff spring’ technique, the freedoms to be fixed are assembled into the global stiffness matrix in the usual way prior to augmenting the appropriate diagonal and force terms (see Section 3.6).

It should also be pointed out that when freedoms are fixed to zero, as is common at the boundaries of solid or structural analyses, the user has the choice of fixing them through boundary condition data, using `nr`, in which case the equations are never assembled, or through the ‘stiff spring’ (or ‘penalty’) technique. The data in Figure 4.12 uses the former strategy, however Figure 4.14 shows an alternative data set that achieves essentially the same results using ‘stiff springs’. An advantage of the ‘stiff spring’ approach for fixing all boundary conditions is that all nodes retain their full complement of freedoms, leading to a simpler freedom numbering system. Disadvantages are that there are greater memory requirements with more equations needing to be solved and the chances of numerical difficulties are higher.

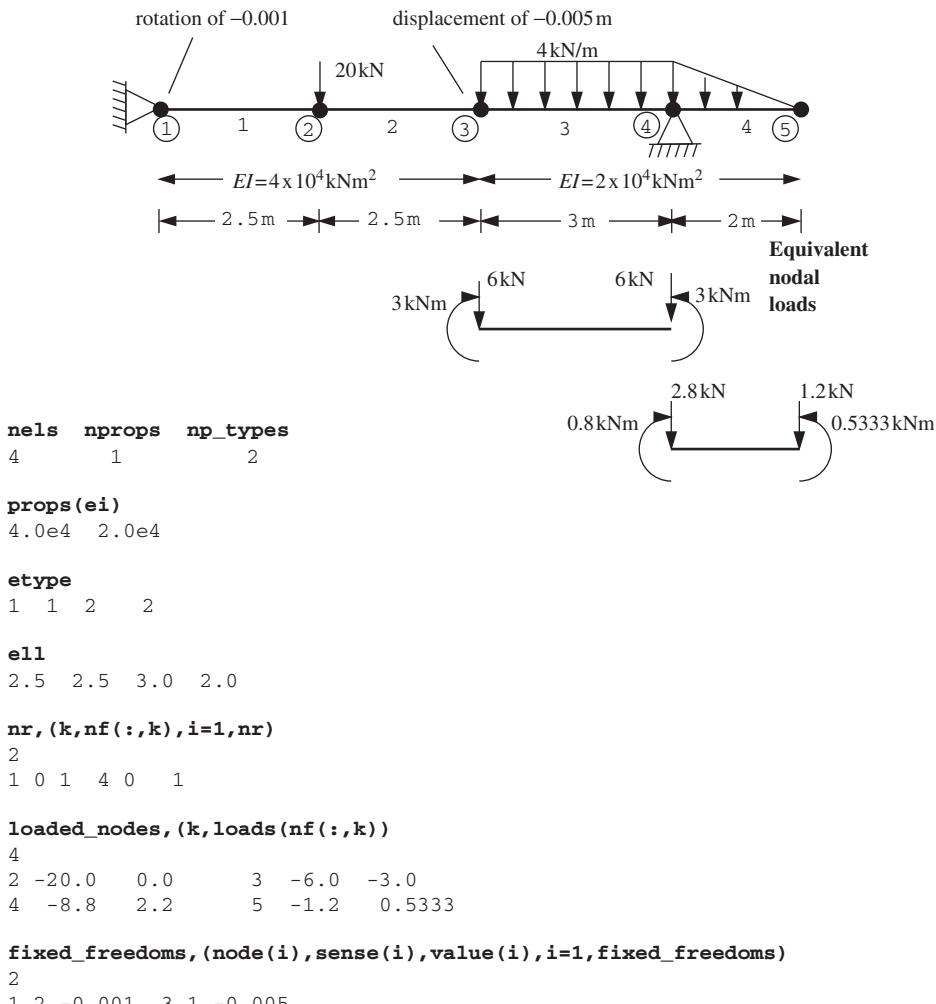


Figure 4.12 Mesh and data for first Program 4.3 example



Figure 4.13 Node and freedom numbering for beam elements

Nodal loading in the context of beam analysis can take the form of either point loads or moments. In the analysis, loading can only be applied at the nodes, so if forces or moments are required between nodes a set of equivalent nodal loads must be derived for application to the mesh. These equivalent nodal loads can be found by computing the shear forces and moments that would have been generated at each node if the element had been fully ‘encastré’ at both ends (the fixed end moments and shear forces). The signs of these fixed end values are then reversed and applied to the nodes of the element in the

```

nels  nprops  np_types
4      1       2

props(ei)
4.0e4  2.0e4

etype
1  1  2  2

ell
2.5  2.5  3.0  2.0

nr
0

loaded_nodes,(k,loads(nf(:,k)))
4
2 -20.0   0.0      3   -6.0   -3.0
4  -8.8    2.2      5   -1.2   0.5333

fixed Freedoms,(node(i),sense(i),value(i),i=1,fixed Freedoms)
4
1 1   0.0      1 2 -0.001
3 1 -0.005   4 1   0.0

```

**Figure 4.14** Alternative data for first Program 4.3 example without using ‘nr’ data

actual problem (see, e.g., Przemieniecki, 1968). If loads or moments are required at the nodes themselves, they are applied directly to the node without any further manipulation. If point loads are to be applied to a beam, it is recommended that a node be placed at that location to avoid the need for ‘fixed end’ calculations.

In the example of Figure 4.12, elements 3 and 4 support distributed loads and the appropriate equivalent nodal loads to be applied are shown beneath their respective elements.

When the computed results shown in Figure 4.15 are examined, it will be seen that the fixed freedoms have the expected values. This is confirmed by the rotation at node 1 of  $-0.001$  (clockwise) and the translation at node 3 of  $-0.005$ . Of the remaining displacements it is seen that the rotation at node 3, for example, equals  $0.000205$  (anti-clockwise).

In order to compute the actual internal shear forces and moments in the beam, the equivalent nodal loads must be subtracted from the corresponding ‘action’ vector printed for each element. This is of course only necessary for those elements that involved

There are 8 equations and the skyline storage is 21

Node	Translation	Rotation
1	0.0000E+00	-0.1000E-02
2	-0.3579E-02	-0.1301E-02
3	-0.5000E-02	0.2051E-03
4	0.0000E+00	0.2410E-02
5	0.4713E-02	0.2343E-02

Element	Force	Moment	Force	Moment
1	0.2157E+02	0.3178E+02	-0.2157E+02	0.2214E+02
2	0.1569E+01	-0.2214E+02	-0.1569E+01	0.2606E+02
3	-0.9577E+01	-0.2906E+02	0.9577E+01	0.3333E+00
4	0.1200E+01	0.1867E+01	-0.1200E+01	0.5333E+00

**Figure 4.15** Results from first Program 4.3 example

loading between the nodes. For example, the shear forces and moments at the nodes in this example for each of the elements are given as follows:

Element 1

$$\begin{aligned} S_1 &= 21.57 \\ M_1 &= 31.78 \\ S_2 &= -21.57 \\ M_2 &= 22.14 \end{aligned}$$

Element 2

$$\begin{aligned} S_1 &= 1.57 \\ M_1 &= -22.14 \\ S_2 &= -1.57 \\ M_2 &= 26.06 \end{aligned}$$

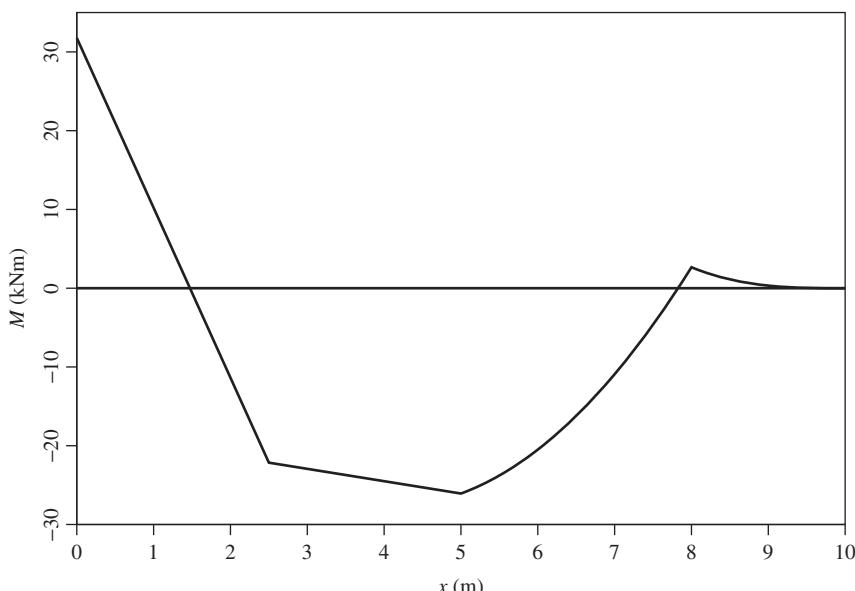
Element 3

$$\begin{aligned} S_1 &= -9.58 + 6.00 = -3.58 \\ M_1 &= -29.06 + 3.00 = -26.06 \\ S_2 &= 9.58 + 6.00 = 15.58 \\ M_2 &= 0.33 - 3.00 = -2.67 \end{aligned}$$

Element 4

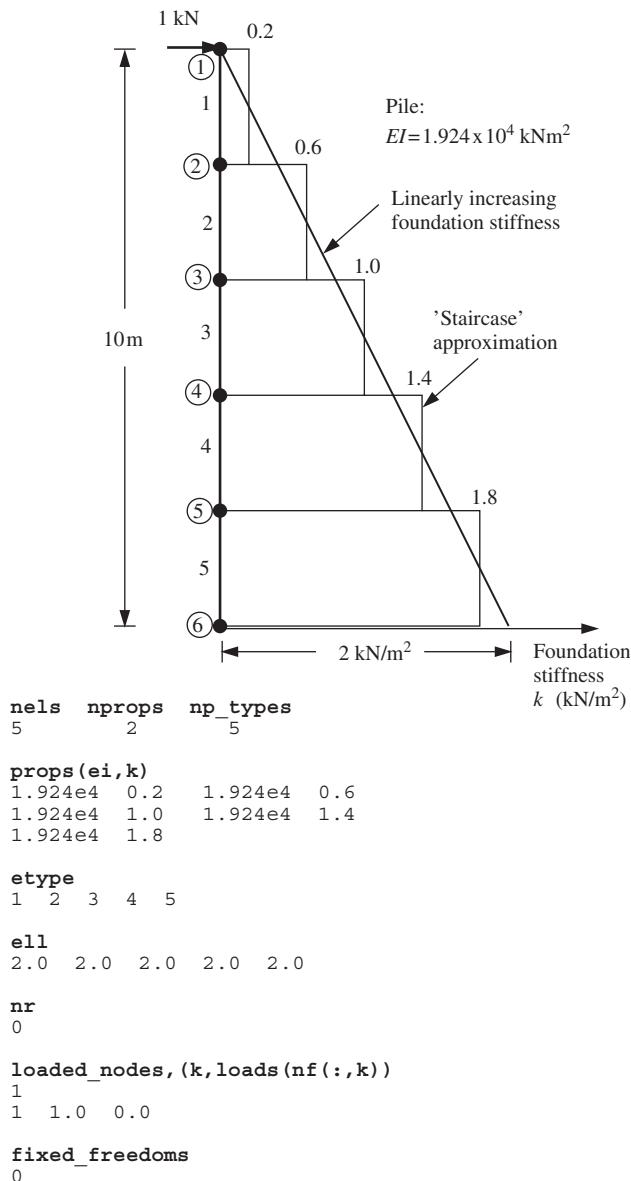
$$\begin{aligned} S_1 &= 12.00 + 2.80 = 14.80 \\ M_1 &= 1.87 + 0.80 = 2.67 \\ S_2 &= -12.00 + 1.20 = -10.80 \\ M_2 &= 0.53 - 0.53 = 0.00 \end{aligned}$$

Internal equilibrium is maintained, for example  $M_2$  of element 2 is equal and opposite to  $M_1$  of element 3, and so on. A bending moment diagram for the beam based on a more refined mesh is given in Figure 4.16.



**Figure 4.16** Bending moment diagram from first Program 4.3 example

A second example of the use of Program 4.3 is illustrated in Figure 4.17 and represents a laterally loaded pile which is to be modelled as a ‘beam on an elastic foundation’, with a linearly increasing soil or foundation stiffness. The pile has a constant flexural stiffness of  $EI = 1.924 \times 10^4 \text{ kNm}^2$ , and the foundation stiffness increases from zero at the ground surface to  $2 \text{ kN/m}^2$  at a depth of 10 m. The pile is modelled by five beam elements, and the soil stiffness is approximated by a step function based on the stiffness at the midpoint of each element. The data file provides `nprops=2` to signify the presence



**Figure 4.17** Mesh and data for second Program 4.3 example

There are 12 equations and the skyline storage is 38

Node	Translation	Rotation
1	0.8559E+00	-0.1148E+00
2	0.6263E+00	-0.1147E+00
3	0.3971E+00	-0.1145E+00
4	0.1683E+00	-0.1143E+00
5	-0.6008E-01	-0.1141E+00
6	-0.2883E+00	-0.1141E+00

Element	Force	Moment	Force	Moment
1	0.1000E+01	0.3638E-11	-0.7036E+00	0.1688E+01
2	0.7036E+00	-0.1688E+01	-0.8960E-01	0.2436E+01
3	0.8960E-01	-0.2436E+01	0.4757E+00	0.1973E+01
4	-0.4757E+00	-0.1973E+01	0.6271E+00	0.7640E+00
5	-0.6271E+00	-0.7640E+00	-0.9095E-12	-0.9095E-12

**Figure 4.18** Results from second Program 4.3 example

of an ‘elastic foundation’, and np\_types=5 since each element is supported by soil with a different stiffness value. The composite beam/foundation global stiffness matrix in this case involves the assembly of the sum of the element stiffness and ‘mass’ matrices km and mm, respectively.

The remainder of the program follows a familiar course. Forces and/or fixed displacements are read, and, following equation solution, the global nodal displacements and rotations are obtained. In the post-processing phase, the element nodal displacements eld are retrieved, as are the element stiffness and ‘mass’ matrices. The product of eld and the element composite stiffness km+mm gives the element ‘actions’ which hold the element shear forces and moments.

The pile is supported by the foundation only, so no additional boundary constraints are needed (nr=0). In this example, a horizontal load of 1.0 kN has been applied to the top of the pile.

The computed results in Figure 4.18 show the horizontal translation and rotation at each of the nodes. It is seen that the horizontal translation of node 1 is given as 0.856. The quite constant rotation computed at each node indicates that the pile is relatively stiff in this example. The reader is invited to repeat the analysis with more elements to demonstrate convergence on the exact solution of 0.902 from Hetenyi (1946).

## Program 4.4 Analysis of elastic rigid-jointed frames using 2-node beam/rod elements in two or three dimensions

```
PROGRAM p44
!-----
! Program 4.4 Analysis of elastic rigid-jointed frames using 2-node
!           beam/rod elements in 2- or 3-dimensions.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim,ndof,nels,neq,nlen,      &
nod=2,nodof,nn,nprops,np_types,nr
REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:,),g_num(:,:,),kdiag(:,),nf(:,:,), &
```

```

no(:,node(:),num(:,sense(:)
REAL(iwp),ALLOCATABLE::action(:),coord(:,:),eld(:,gamma(:,g_coord(:, :, &
km(:,:,kv(:,loads(:,prop(:,value(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,nn,ndim,nprops,np_types
IF(ndim==2)nodof=3; IF(ndim==3)nodof=6; ndof=nod*nodof
ALLOCATE(nf(nodof,nn),km(ndof,ndof),coord(nod,ndim),g_coord(ndim,nn), &
eld(ndof),action(ndof),g_num(nod,nels),num(nod),g(ndof),gamma(nels), &
g_g(ndof,nels),prop(nprops,np_types),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype;
IF(ndim==3)READ(10,*)gamma
READ(10,*)g_coord; READ(10,*)g_num
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
  num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
  CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))') &
  " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global stiffness matrix assembly-----
kv=zero
elements_2: DO iel=1,nels
  num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
  CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
  CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read loads and/or displacements-----
loads=zeros; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
  ALLOCATE(node(fixed_freedoms),no(fixed_freedoms), &
    sense(fixed_freedoms),value(fixed_freedoms))
  READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
  DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
  kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution -----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
WRITE(11,'(/A)') " Node Displacements and Rotation(s)"
DO k=1,nn; WRITE(11,'(I5,6E12.4)')k,loads(nf(:,k)); END DO
!-----retrieve element end actions-----
WRITE(11,'(/A)') " Element Actions"
elements_3: DO iel=1,nels
  num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
  CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
  eld=loads(g); action=MATMUL(km,eld)
  IF(ndim<3)THEN; WRITE(11,'(I5,6E12.4)')iel,action; ELSE
    WRITE(11,'(I5,6E12.4)')iel, action(1: 6)
    WRITE(11,'(A,6E12.4)')" ",action(7:12)
  END IF
END DO elements_3
STOP
END PROGRAM p44

```

The first three programs in this chapter were concerned only with 1D elements which could sustain either axial loads (rod elements) or transverse loading and moments (beam elements). It is much more common to encounter structures made up of members arbitrarily inclined and attached to one another. Loading of such structures results in displacements due to both axial and bending effects, although the former is often ignored in many approximate methods. The beam–rod element stiffness matrix used by this program is formed by superposing the beam and rod stiffness matrices described in earlier programs in this chapter. The program described in this section can analyse 2D or 3D framed structures, with the element stiffness matrix formed by the library subroutine `rigid_jointed`.

The first example analysed by Program 4.4 is shown in Figure 4.19 and is a 2D rigid-jointed frame subjected to distributed loads and point loads. In 2D, the elements have six degrees of freedom as shown in Figure 4.20. At each node there are two translational freedoms in  $x$  and  $y$  and a rotation (in that order). The nodal freedom numbering associated with each element, accounting for any restraints, is as usual contained in the ‘steering’ vector  $\mathbf{g}$ . Thus, with reference to Figures 4.19 and 4.20, the steering vector for element 1 would be  $[0\ 0\ 1\ 2\ 3\ 4]^T$  and for element 2,  $[2\ 3\ 4\ 5\ 6\ 7]^T$ , and so on. The data organisation is similar to the pin-jointed frame analysis described in Program 4.2. The first line of data provides the number of elements (`nels`), the number of nodes (`nn`), the dimensionality (`ndim`), the number of properties `nprops` and the number of property types `np_types`.

In this program, the number of material properties required depends on the dimensionality, so the data now includes input to the integer `nprops` which indicates the number of material properties required for each property type. In a 2D frame problem, there are two material properties required (`EA` and `EI`), so `nprops = 2`. The material property values for each type are then read into the two-dimensional array `prop`.

The material property data is followed by the `etype` vector (if needed), the global nodal coordinates (`g_coord`) and the element node numbering (`g_num`). The loading on the nodes is calculated using the equivalent nodal loads approach described previously for the first example with Program 4.3, and these values are shown for each individual element in Figure 4.19. There are no `fixed_freedoms` in this example.

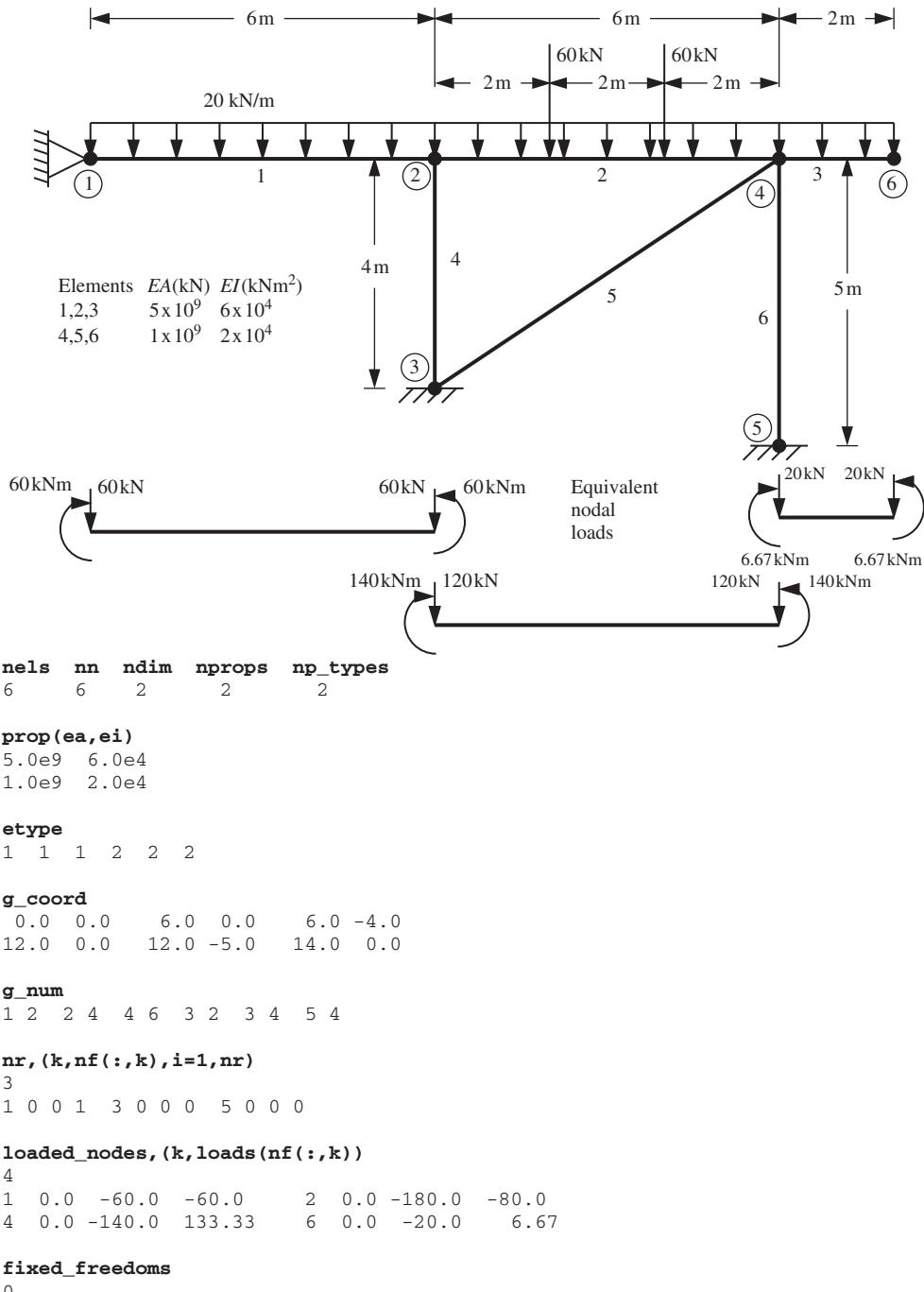
The results shown in Figure 4.21 indicate that the rotation at node 1, for example, is  $-0.001025$  (clockwise). The action vectors for elements 4, 5 and 6 are correct as printed, however for elements 1, 2 and 3 the equivalent nodal loads must be subtracted:

#### Element 1

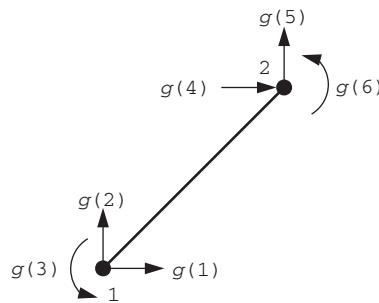
$$\begin{aligned} F_{x1} &= -30.38 + 0.00 = -30.38 \\ F_{y1} &= -19.75 + 60.00 = 40.25 \\ M_1 &= -60.00 + 60.00 = 0.00 \\ F_{x1} &= 30.38 + 0.00 = 30.38 \\ F_{y1} &= 19.75 + 60.00 = 79.75 \\ M_2 &= -58.49 - 60.00 = -118.49 \end{aligned}$$

#### Element 2

$$\begin{aligned} F_{x1} &= -23.25 + 0.00 = -23.25 \\ F_{y1} &= 8.24 + 120.00 = 128.24 \\ M_1 &= -2.52 + 140.00 = 137.48 \\ F_{x1} &= 23.25 + 0.00 = 23.25 \\ F_{y1} &= -8.24 + 120.00 = 111.76 \\ M_2 &= 51.95 - 140.00 = -88.05 \end{aligned}$$



**Figure 4.19** Mesh and data for first Program 4.4 example



**Figure 4.20** Node and freedom numbering for 2D beam–rod elements

There are 10 equations and the skyline storage is 40

Node	Displacements and Rotation(s)		
1	0.0000E+00	0.0000E+00	-0.1025E-02
2	0.3645E-07	-0.8319E-06	-0.9497E-03
3	0.0000E+00	0.0000E+00	0.0000E+00
4	0.6435E-07	-0.6283E-06	0.1774E-02
5	0.0000E+00	0.0000E+00	0.0000E+00
6	0.6435E-07	0.2880E-02	0.1329E-02

#### Element Actions

1	-0.3038E+02	-0.1975E+02	-0.6000E+02	0.3038E+02	0.1975E+02	-0.5849E+02
2	-0.2325E+02	0.8238E+01	-0.2519E+01	0.2325E+02	-0.8238E+01	0.5195E+02
3	0.0000E+00	0.2000E+02	0.3333E+02	0.0000E+00	-0.2000E+02	0.6670E+01
4	0.7123E+01	0.2080E+03	-0.9497E+01	-0.7123E+01	-0.2080E+03	-0.1899E+02
5	0.3177E+02	0.2610E+02	0.9839E+01	-0.3177E+02	-0.2610E+02	0.1968E+02
6	-0.8513E+01	0.1257E+03	0.1419E+02	0.8513E+01	-0.1257E+03	0.2838E+02

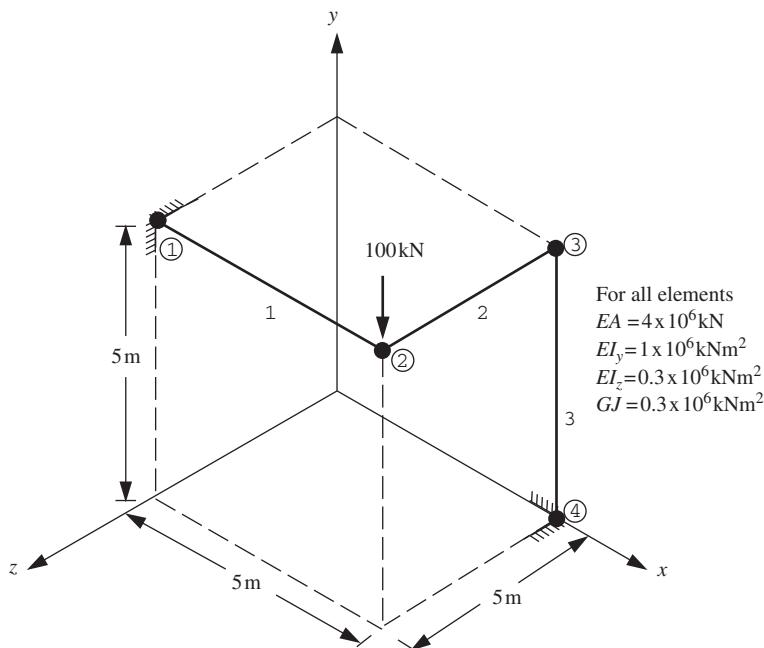
**Figure 4.21** Results from first Program 4.4 example

#### Element 3

$$\begin{aligned}
 F_{x1} &= 0.00 + 0.00 = 0.00 \\
 F_{y1} &= 20.00 + 20.00 = 40.00 \\
 M_1 &= 33.33 + 6.67 = 40.00 \\
 F_{x1} &= 0.00 + 0.00 = 0.00 \\
 F_{y1} &= -20.00 + 20.00 = 0.00 \\
 M_2 &= 6.67 - 6.67 = 0.00
 \end{aligned}$$

Moment equilibrium is established by adding the moments from the appropriate end of all elements coming into the joint.

The second example to be analysed by Program 4.4 is shown in Figure 4.22 and represents a 3D rigid-jointed frame subjected to a vertical point load of -100.0. In 3D, the elements have 12 degrees of freedom as shown in Figure 4.23. At each node there are three translational freedoms in  $x$ ,  $y$  and  $z$ , and three rotations about each of the global axes. The extension to 3D is conceptually simple, but considerably more care is required in the preparation of data and attention to sign conventions. The data organisation is virtually the same as in the previous example, except `ndim` is set to 3 in the data.



```

nels    nn    ndim   nprops   np_types
3        4      3        4          1

prop(ea,eiy,eiz,gj)
4.0e6  1.0e6  0.3e6  0.3e6

etypenot needed)

gamma
0.0    0.0    90.0

g_coord
0.0    5.0    5.0      5.0    5.0    5.0
5.0    5.0    0.0      5.0    0.0    0.0

g_num
1 2 3 2 4 3

nr, (k, nf(:,k), i=1, nr)
2
1 0 0 0 0 0 0 4 0 0 0 0 0 0

loaded_nodes, (k, loads(nf(:,k)))
1
2 0.0 -100.0 0.0 0.0 0.0 0.0

fixed_freedoms
0

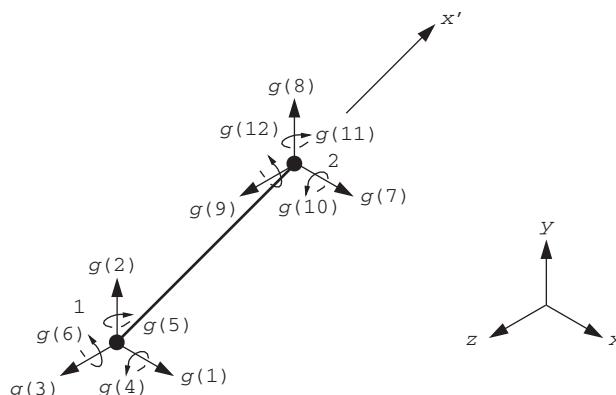
```

Figure 4.22 Mesh and data for second Program 4.4 example

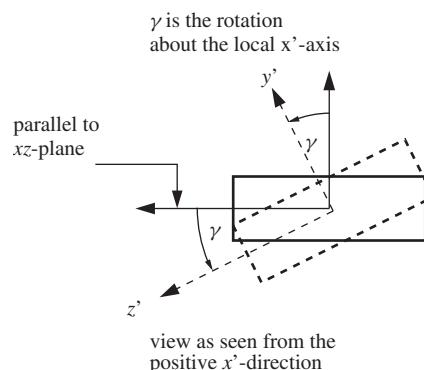
In addition to the axial stiffness ( $EA$ ), 3D involves the flexural stiffness about the element's local  $y'$  and  $z'$  axes ( $EI_y$  and  $EI_z$ , respectively) and a torsional stiffness ( $GJ$ ), thus  $nprops$  is 4. The local coordinate  $x'$  defines the long axis of the element. The relationship between the global axes ( $x, y, z$ ) and local axes ( $x', y', z'$ ) must be considered for 3D space frames because in addition to the six coordinates that define the position of each node of the element in space, a seventh rotational 'coordinate'  $\gamma$  must be read in as data. The additional real vector gamma is provided to hold this information (in degrees) for each element.

For the purposes of data preparation, a 'vertical' element is defined as one which lies parallel to the global  $y$ -axis. For non-vertical elements, the angle  $\gamma$  is defined as the rotation of the element about its local  $x'$ -axis as shown in Figure 4.24. For 'vertical' elements, however,  $\gamma$  is defined as the angle between the the global  $z$ -axis and the local  $z'$ -axis, measured towards the global  $x$ -axis as shown in Figure 4.25. For 'vertical' elements it is essential that the local  $x'$ -axis points in the same direction as the global  $y$ -axis.

Returning to the example problem, it is necessary to establish the local coordinate system for each element as shown in Figure 4.26. As indicated in Figure 4.23, the positive



**Figure 4.23** Node and freedom numbering for 3D beam–rod elements



**Figure 4.24** Transformation angle for 'non-vertical' elements

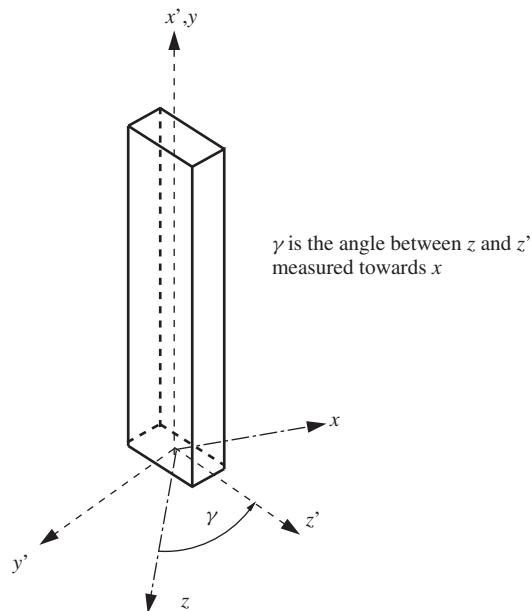


Figure 4.25 Transformation angle for ‘vertical’ elements

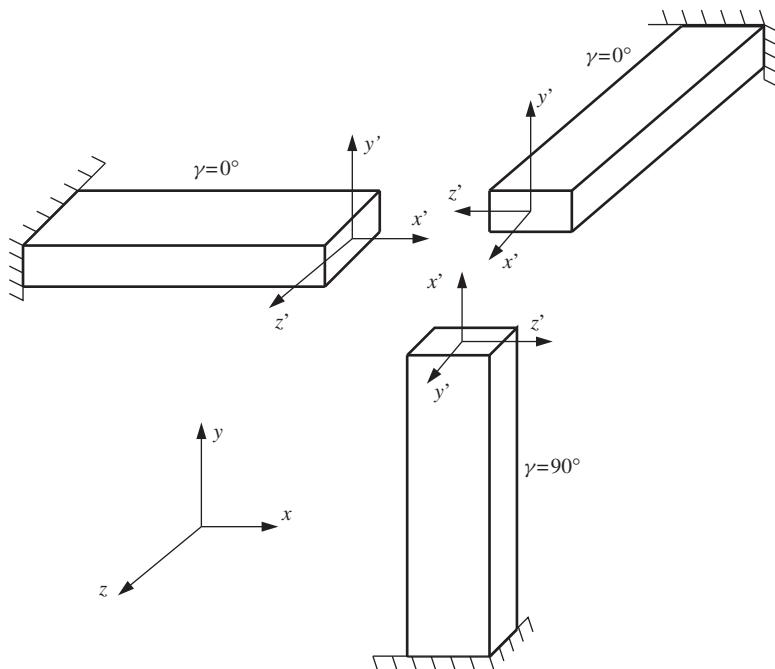


Figure 4.26 Element local coordinate systems

There are 12 equations and the skyline storage is 78

Node Displacements and Rotation(s)							
1	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
2	-0.3039E-05	-0.5997E-02	0.8769E-03	0.1129E-02	-0.2360E-03	-0.1514E-02	
3	0.9571E-03	-0.4536E-04	0.9113E-03	0.7470E-03	-0.1582E-03	-0.3727E-03	
4	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00

Element Actions							
1	0.2431E+01	0.6371E+02	-0.2754E+02	-0.6777E+02	0.1160E+03	0.2501E+03	
	-0.2431E+01	-0.6371E+02	0.2754E+02	0.6777E+02	0.2165E+02	0.6846E+02	
2	-0.2431E+01	0.3629E+02	0.2754E+02	-0.1137E+03	0.9490E+01	0.6846E+02	
	0.2431E+01	-0.3629E+02	-0.2754E+02	-0.6777E+02	-0.2165E+02	-0.6846E+02	
3	-0.2431E+01	0.3629E+02	0.2754E+02	0.2403E+02	0.9490E+01	0.8062E+02	
	0.2431E+01	-0.3629E+02	-0.2754E+02	0.1137E+03	-0.9490E+01	-0.6846E+02	

**Figure 4.27** Results from second Program 4.4 example

local  $x'$ -direction is defined by moving from node 1 to node 2, so this is also the order in which the element nodal numbering must be given in the data.

In the example of Figure 4.22, elements 1 and 2 both have their local  $z'$ -axes parallel to the global  $xz$ -plane, thus there has been no rotation of these non-vertical elements and  $\gamma$  is set to zero. For vertical element 3, however,  $\gamma$  is set to  $90^\circ$ , which is the angle between the global  $z$ - and local  $z'$ -axes measured towards the global  $x$ -axis.

The results of the analysis given in Figure 4.27 indicate that the vertical deflection under the load is  $-0.005997$ . As a check on equilibrium under the applied loading of  $-100.0$ , the  $F_{y1}$  component of the action vector at the built-in end (node 1) of each of the elements 1 and 3 is 63.71 and 39.29, respectively.

## Program 4.5 Analysis of elastic-plastic beams or frames using 2-node beam or beam/rod elements in one, two or three dimensions

```

PROGRAM p45
!-----
! Program 4.5 Analysis of elasto-plastic beams or rigid-jointed frames
!           using 2-node beam or beam/rod elements in 1-, 2- or
!           3-dimensions
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,incs,iters,iy,k,limit,loaded_nodes,ndim,ndof,nels,neq,      &
nlen,nod=2,nodof,nn,nprops,np_types,nr;; CHARACTER(LEN=15)::argv
REAL(iwp)::tol,total_load,zero=0.0_iwp; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:,),g_g(:,,:),g_num(:,,:),kdiag(:,),nf(:, :,), &
node(:,),num(:,)
REAL(iwp),ALLOCATABLE::action(:,),bdylds(:,),coord(:, :,),dload(:,),eld(:,),    &
elddtot(:,),gamma(:,),g_coord(:, :,),holdr(:, :,),km(:, :,),kv(:, ),loads(:, ),&
oldis(:, ),prop(:, :,),react(:, ),val(:, :))
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,nn,ndim,nprops,np_types
IF(ndim==1)nodof=2; IF(ndim==2)nodof=3; IF(ndim==3)nodof=6; ndof=nod*nodof
ALLOCATE(nf(nodof,nn),km(ndof,ndof),coord(nod,ndim),g_coord(ndim,nn),      &
```

```

eld(ndof),action(ndof),g_num(nod,nels),num(nod),g(ndof),gamma(nels),      &
g_g(ndof,nels),holdr(ndof,nels),react(ndof),prop(nprops,np_types),      &
etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype;
IF(ndim==3)READ(10,*)gamma
READ(10,*)g_coord; READ(10,*)g_num
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),eldtot(0:neq),bdylds(0:neq),oldis(0:neq))
!-----loop the elements to find global array sizes-----
kdiag=0
elements_1: DO iel=1,nels
    num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')                                     &
" There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global stiffness matrix assembly-----
holdr=zero; kv=zero
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
    CALL fsparv(kv,km,g,kdiag)
END DO elements_2
READ(10,*)loaded_nodes
ALLOCATE(node(loaded_nodes),val(loaded_nodes,nodof))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes); READ(10,*)limit,tol,incs
ALLOCATE(dload(incs)); READ(10,*)dload
!-----equation factorisation-----
CALL sparin(kv,kdiag); total_load=zero
!-----load increment loop-----
load_incs: DO iy=1,incs
    total_load=total_load+dload(iy); WRITE(*,'(/,A,I5)')" load step",iy
    WRITE(11,'(/A,i3,A,E12.4)')" Load step",iy," Load factor ",total_load
    oldis=zero; iters=0
    its: DO
        iters=iters+1; WRITE(*,"iteration no",iters, loads=zero
        DO i=1,loaded_nodes; loads(nf(:,node(i)))=dload(iy)*val(i,:); END DO
        loads=loads+bdylds; bdylds=zero
    !-----forward/back-substitution and check convergence---
        CALL spabac(kv,loads,kdiag); loads(0)=zero
        CALL checon(loads,oldis,tol,converged)
    !-----inspect moments in all elements-----
    elements_3: DO iel=1,nels
        num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
        CALL rigid_jointed(km,prop,gamma,etype,iel,coord); g=g_g(:,iel)
        eld=loads(g); action=MATMUL(km,eld); react=zero
    !-----if plastic moments exceeded generate correction vector-
        IF(limit/=1)THEN
            CALL hinge(coord,holdr,action,react,prop,iel,etype,gamma)
            bdylds(g)=bdylds(g)-react; bdylds(0)=zero
        END IF
    !-----at convergence update element reactions-----
        IF(iters==limit.OR.converged)                                     &
            holdr(:,iel)=holdr(:,iel)+react(:)+action(:)
    END DO elements_3
    IF(iters==limit.OR.converged)EXIT its
END DO its

```

```

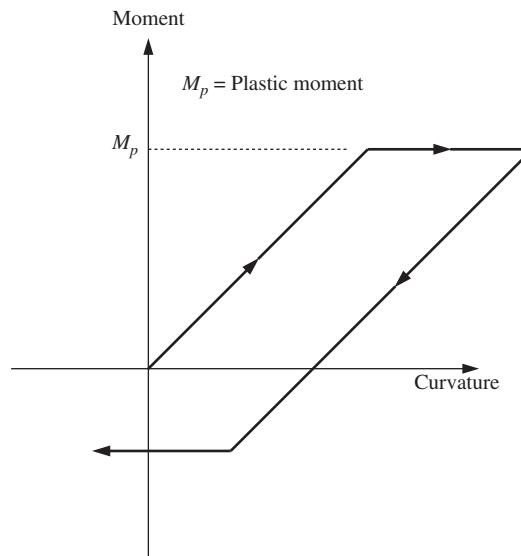
eldtot=loads+eldtot;
WRITE(11,'(A)')    " Node    Displacement(s) and Rotation(s)"
DO i=1,loaded_nodes
  WRITE(11,'(I5,6E12.4)')node(i),eldtot(nf(:,node(i)))
END DO
WRITE(11,'(A,I5,A)')" Converged in",iters," iterations"
IF(iters==limit.AND.limit/=1)EXIT load_incs
END DO load_incs
STOP
END PROGRAM p45

```

This program is an extension of the preceding Program 4.4 in which a limit is placed on the maximum moment that any member can sustain. As loads on the structure are increased, plastic hinges form progressively and ‘collapse’ occurs when a mechanism develops. This program is currently limited to load control analysis.

This is the first example in the book of a non-linear analysis in which the moment–curvature behaviour of the members is assumed to be elastic–perfectly plastic as shown in Figure 4.28. To deal with this non-linearity, an iterative approach is used to find the nodal displacements and element ‘actions’ under a given set of applied loads. Moments in excess of their plastic limits are redistributed to other joints which still have reserves of moment carrying capacity. Convergence of the iterative process is said to have occurred when, within tolerances, moments at the element nodes nowhere exceed their limiting plastic values, and the internal ‘actions’ are in equilibrium with the applied external loads.

The conventional approach for tackling this type of problem is progressively to modify the global stiffness matrix as joints reach the plastic limit. The modification is necessary because a plastic joint is replaced by a pin joint with the appropriate plastic moment

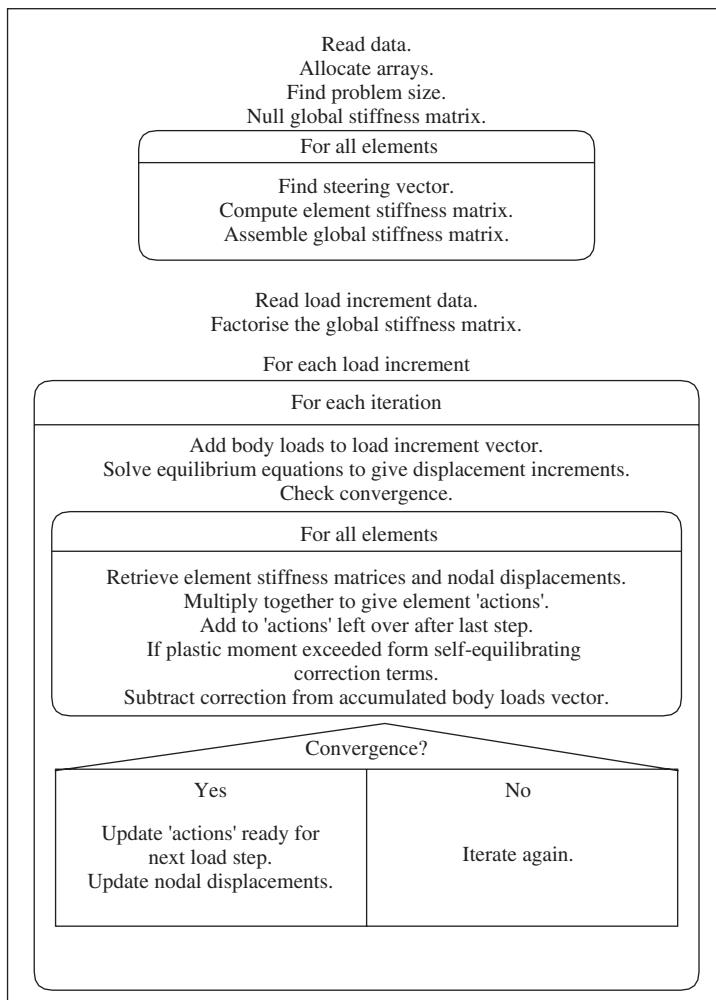


**Figure 4.28** Elastic–perfectly plastic moment–curvature relationship

applied. In the method shown in the structure chart in Figure 4.29, the global stiffness matrix is formed once only, with the non-linearity introduced by iteratively modifying the applied forces on the structure until convergence is achieved. For greater detail of this particular algorithm, the reader is referred to Griffiths (1988). Similar procedures are utilised in Chapter 6 in relation to elastic–plastic solids.

When a problem involves a constant left-hand-side matrix and multiple right-hand-side vectors, the benefits of splitting the solution of the equilibrium equations into two stages, namely factorisation performed once (using subroutine `sparin`), and a forward and back-substitution performed at each iteration for each new right-hand side (using subroutine `spabac`) become clear.

Material properties in Program 4.5 must now include both elastic properties and plastic moment values for all members. For 1D beams or 2D frames, only one plastic moment



**Figure 4.29** Structure chart for Program 4.5

$(M_p)$  is read, thus nprops is 2 in 1D and 3 in 2D. For 3D space frames, however, three plastic moments ( $M_{py}$ ,  $M_{pz}$  and  $M_{px}$  in that order) are read, thus nprops is 7, where  $M_{py}$  and  $M_{pz}$  represent, respectively, the limiting bending moment about the local  $y'$ - and  $z'$ -axes of the member, and  $M_{px}$  represents the limiting torsional moment about the long axis of the member.

Loads are applied in `incs` increments to the nodes and the magnitude of each increment is read into the vector `dload`. The loading remains proportional as is customary in plastic hinge analysis, so the relative magnitudes of the nodal loads are read by `node` and `val`, where `node` holds the node numbers and `val` holds the load weightings on each freedom.

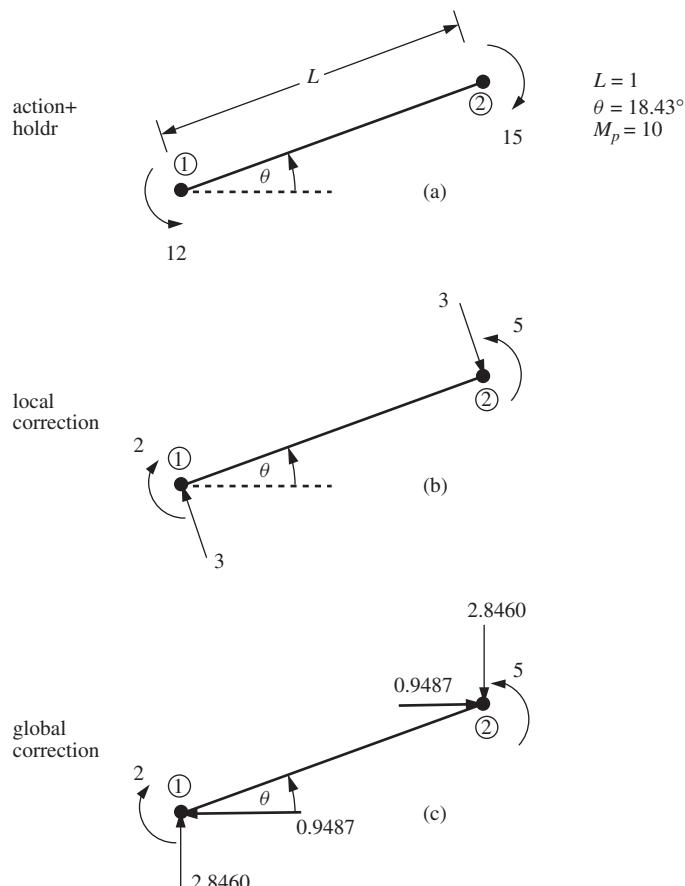
Following assembly of the global stiffness matrix and factorisation by subroutine `sparin`, the program enters the load increment loop. For each iteration counted by `iters`, the external load increments are added to the redistributive loads vector `bddylds`. The equilibrium equations are solved using subroutine `spabac` and the resulting nodal displacement increments compared with their values at the previous iteration using subroutine `checon`. This subroutine observes the relative change in displacement increments from one iteration to the next. If the change is less than `tol` then the logical variable `converged` is set to `.TRUE.` and convergence has occurred. Alternatively, `converged` is set to `.FALSE.` and another iteration is performed.

At each iteration, each element is inspected and its action vector computed from the product of its nodal displacements and the element stiffness matrix. The subroutine `hinge` adds the action vector to the values already existing from the previous load step (held in `holdr`) and checks both nodes to see if the plastic moment value has been exceeded. If the plastic moment value has been exceeded, the self-equilibrating vector `react` is formed. In Figure 4.30(a), a typical 2D element is shown in which a particular load increment has pushed the moment value at both nodes over their plastic limit. The correction vector applies a moment to each node equal to the amount of overshoot of the plastic moment values, however, to preserve equilibrium, a couple is required as shown in the local coordinate system in Figure 4.30(b). Finally, as shown in Figure 4.30(c), the couple is transformed into global coordinate directions before being assembled into the `bddylds` vector. Only those elements that have moments in excess of the plastic limits will contribute any loading to `bddylds`.

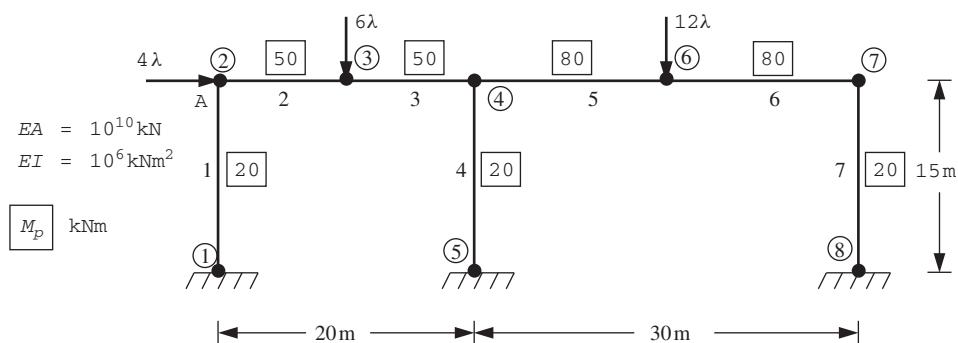
If, at any load step, the algorithm fails to converge within the prescribed iteration ceiling `limit` then ‘collapse’ of the structure is indicated, because the algorithm has been unable to satisfy equilibrium without violating the plastic moment values.

The first example shown in Figure 4.31 is a two-bay portal frame subjected to proportional loading. After each increment, the output shown in Figure 4.32 gives the loading factor  $\lambda$  (equal to the accumulated values of `dload`) together with the loaded nodal displacements and the iteration count to achieve convergence. To reduce the volume of output, only the displacements of loaded nodes (2, 3 and 6) are given. In Figure 4.33, the horizontal movement of point A is plotted against  $\lambda$ , indicating close agreement with the theoretical value of  $\lambda_f = 1.375$  given by Horne (1971) for this problem.

The second example in 3D shown in Figure 4.34 is of a plane triangular grid, rigidly supported along one side and subjected to a point transverse load at its apex (node 7). All members have the same stiffness and plastic moment. The output shown in Figure 4.35 indicates that failure occurs when the transverse load approaches a value of 1.55.



**Figure 4.30** Correction terms for ‘yielding’ element



**Figure 4.31** Mesh and data for first Program 4.5 example (*Continued*)

```

nels nn ndim nprops np_types
7     8    2      3      3

prop(ea,ei,mp)
1.0e10   1.0e6  20.0
1.0e10   1.0e6  50.0
1.0e10   1.0e6  80.0

etype
1 2 2 1 3 3 1

g_coord
0.0 0.0    0.0 15.0   10.0 15.0   20.0 15.0
20.0 0.0   35.0 15.0   50.0 15.0   50.0 0.0

g_num
1 2 2 3 3 4 5 4 4 6 6 7 8 7

nr, (k,nf(:,k),i=1,nr)
3
1 0 0 0 5 0 0 0 8 0 0 0

loaded_nodes, (node(i),val(i,:),i=1,loaded_nodes)
3
2 4.0    0.0    0.0
3 0.0    -6.0   0.0
6 0.0    -12.0  0.0

limit tol
200 0.0001

incs,(dload(i),i=1,incs)
8
0.5 0.3 0.2 0.2 0.1 0.05 0.02 0.01

```

**Figure 4.31 (Continued)**

There are 15 equations and the skyline storage is 66

```

Load step 1 Load factor 0.5000E+00
Node Displacement(s) and Rotation(s)
 2 0.2073E-03 -0.9812E-09 -0.2015E-04
 3 0.2073E-03 -0.8478E-04  0.1410E-04
 6 0.2073E-03 -0.1161E-02 -0.3057E-05
Converged in 2 iterations

Load step 2 Load factor 0.8000E+00
Node Displacement(s) and Rotation(s)
 2 0.4912E-03 -0.1102E-08 -0.3817E-04
 3 0.4912E-03 -0.1133E-03  0.2775E-04
 6 0.4912E-03 -0.2210E-02 -0.2088E-04
Converged in 25 iterations

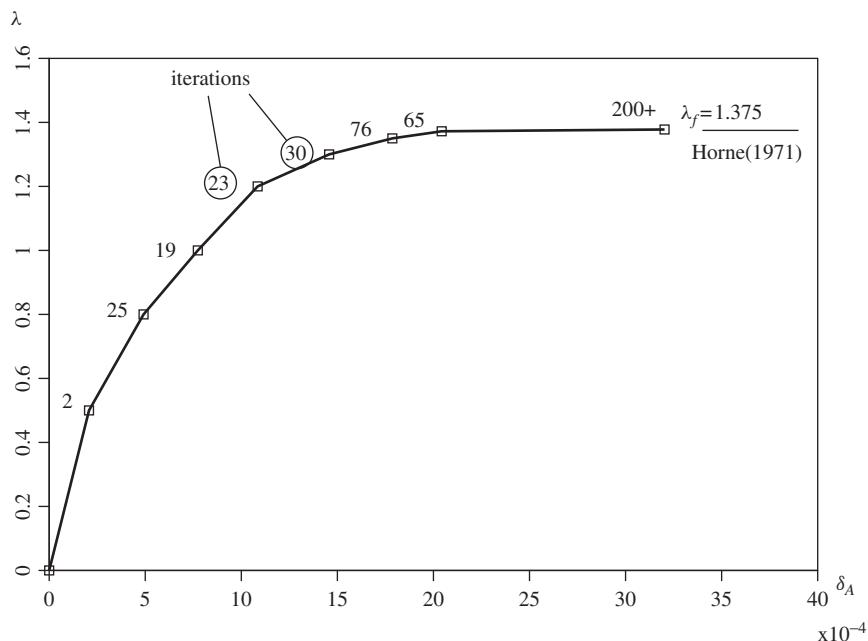
.
.

Load step 7 Load factor 0.1370E+01
Node Displacement(s) and Rotation(s)
 2 0.2043E-02 -0.9284E-09 -0.1300E-03
 3 0.2043E-02 -0.2059E-03  0.9912E-04
 6 0.2043E-02 -0.5453E-02 -0.5822E-04
Converged in 65 iterations

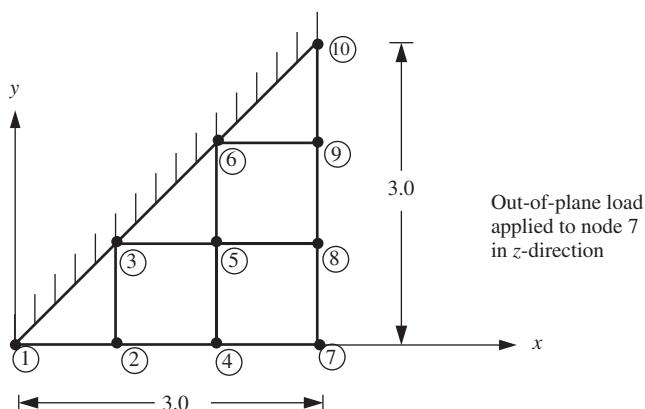
Load step 8 Load factor 0.1380E+01
Node Displacement(s) and Rotation(s)
 2 0.3203E-02 -0.9416E-09 -0.1087E-03
 3 0.3203E-02  0.2461E-04  0.1241E-03
 6 0.3203E-02 -0.6727E-02 -0.5976E-04
Converged in 200 iterations

```

**Figure 4.32** Results from first Program 4.5 example



**Figure 4.33** Load displacement behaviour from first Program 4.5 example



**Figure 4.34** Mesh and data for second Program 4.5 example (*Continued*)

```

nels   nn   ndim   nprops   np_types
12     10     3      7         1

prop(ea,eiy,eiz,gj,mpy,mpz,mpx)
1.0  1.e4  1.e4  1.0  1.0  1.0  1.e8

etype(not needed)

gamma
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

g_coord
0.0  0.0  0.0    1.0  0.0  0.0    1.0  1.0  0.0    2.0  0.0  0.0
2.0  1.0  0.0    2.0  2.0  0.0    3.0  0.0  0.0    3.0  1.0  0.0
3.0  2.0  0.0    3.0  3.0  0.0

g_num
1 2  2 3  2 4  3 5  4 5  5 6  4 7  5 8  6 9  7 8  8 9  9 10

nr,(k,nf(:,k),i=1,nr)
4
1 0 0 0 0 0 0 3 0 0 0 0 0 0 6 0 0 0 0 0 0 10 0 0 0 0 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
1
7 0.0  0.0  1.0  0.0  0.0  0.0

limit,tol
200  0.00001

incs,(dload(i)=1,incs)
5
0.5 0.5 0.5 0.05 0.05

```

**Figure 4.34** (Continued)

There are 36 equations and the skyline storage is 378

```

Load step 1 Load factor 0.5000E+00
Node Displacement(s) and Rotation(s)
 7 0.0000E+00 0.0000E+00 0.1059E-03 -0.6345E-04 -0.6345E-04 0.0000E+00
Converged in 3 iterations

Load step 2 Load factor 0.1000E+01
Node Displacement(s) and Rotation(s)
 7 0.0000E+00 0.0000E+00 0.2117E-03 -0.1269E-03 -0.1269E-03 0.0000E+00
Converged in 3 iterations

Load step 3 Load factor 0.1500E+01
Node Displacement(s) and Rotation(s)
 7 0.0000E+00 0.0000E+00 0.3247E-03 -0.1936E-03 -0.1936E-03 0.0000E+00
Converged in 18 iterations

Load step 4 Load factor 0.1550E+01
Node Displacement(s) and Rotation(s)
 7 0.0000E+00 0.0000E+00 0.1517E-02 -0.7944E-03 -0.7944E-03 0.0000E+00
Converged in 200 iterations

```

**Figure 4.35** Results from second Program 4.5 example

## Program 4.6 Stability (buckling) analysis of elastic beams using 2-node beam elements (elastic foundation optional)

```

PROGRAM p46
!-----
! Program 4.6 Stability (buckling) analysis of elastic beams using 2-node
!           beam elements (elastic foundation optional).
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,iters,k,limit,ndof=4,nels,neq,nlen,nod=2,nodof=2,nn,
nprops,np_types,nr
REAL(iwp)::eval,tol,zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,),g(:,),g_g(:,:),kdiag(:,),nf(:,,:),num(:,)
REAL(iwp),ALLOCATABLE::ell(:,),evec(:,),gm(:,,:),gv(:,),km(:,,:),kv(:,),
mm(:,,:),prop(:,,:))
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,nprops,np_types; nn=nels+1
ALLOCATE(nf(nodof,nn),ell(nels),num(nod),g(ndof),g_g(ndof,nels),
etype(nels),prop(nprops,np_types),km(ndof,ndof),gm(ndof,ndof),
mm(ndof,ndof))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr),limit,tol; CALL formnf(nf)
neq=MAXVAL(nf); ALLOCATE(kdiag(neq),evec(0:neq)); kdiag=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),gv(kdiag(neq)))
WRITE(11,'(2(A,I5))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global stiffness and geometric matrix assembly-----
kv=zero; gv=zero
elements_2: DO iel=1, nels
    CALL beam_km(km,prop(1,etype(iel)),ell(iel)); g=g_g(:,iel)
    mm=zero; IF(nprops>1)CALL beam_mm(mm,prop(2,etype(iel)),ell(iel))
    CALL beam_gm(gm,ell(iel))
    CALL fsparv(kv,km+mm,g,kdiag); CALL fsparv(gv,gm,g,kdiag)
END DO elements_2
!-----solve eigenvalue problems-----
CALL stability(kv,gv,kdiag,tol,limit,iters,evec,eval)
WRITE(11,'(/A,E12.4,/)')" The buckling load =",eval
evec(0)=zero
WRITE(11,'(A)')" The buckling mode"
WRITE(11,'(/A)')" Node Translation Rotation"
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,evec(nf(:,k)); END DO
WRITE(11,'(/A,I5,A)')" Converged in",iters," iterations"
STOP
END PROGRAM p46

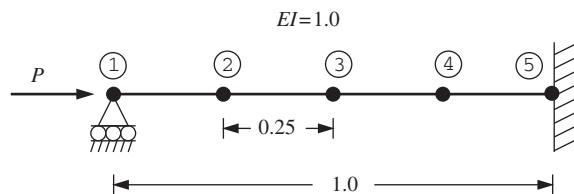
```

This program computes the fundamental (lowest) buckling load and associated mode shape of beam elements subjected to axial compressive loading. The program has much in common with Program 4.3 and includes the option of an elastic foundation. The element geometric matrices `gm` (see Section 2.5) are generated by subroutine `beam_gm` and the assembled geometric matrix stored in vector `gv`. The beam/foundation stiffness matrices are `km` and `mm`, and assembled into `kv` as in Program 4.3.

Subroutine `stability` uses simple iteration to solve the eigenvalue problem for the smallest eigenvalue called `eval`, corresponding to the lowest buckling load. The program prints the buckling load and the corresponding eigenvector `evec`, or mode shape, scaled so that its Euclidean norm equals unity. Other variables in this program relating to the eigenvalue solver include `tol`, which is the convergence tolerance for the iterative algorithm, `iters`, which holds the number of iterations to achieve convergence and `limit`, which represents the iteration ceiling.

The first example shown in Figure 4.36 is of a beam of unit length, fixed at one end and pinned at the other. Since only a beam is being analysed, `nprops=1`. Four beam elements, each of length 0.25, have been used to model the beam, and for simplicity, the flexural stiffness  $EI$  has also been set to unity. The output shown in Figure 4.37 gives a buckling load of 20.23, which agrees closely with the theoretical solution of  $2.04\pi^2EI/L^2 = 20.19$ .

The second example shown in Figure 4.38 is of a simply supported beam on an elastic foundation. As in the previous example, four elements have been used to discretise the problem, with the foundation stiffness set to 800.0. The result given in Figure 4.39



```

nels    nprops   np_types
4        1         1

prop (ei)
1.0

etype(not needed)

e11
0.25 0.25 0.25 0.25

nr, (k,nf(:,k),i=1,nr)
2
1 0 1 5 0 0

limit      tol
100       1.0e-5

```

**Figure 4.36** Mesh and data for first Program 4.6 example

There are 7 equations and the skyline storage is 20

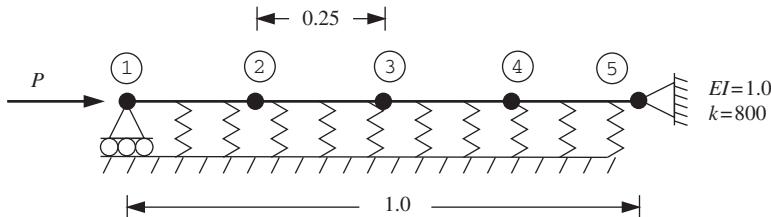
The buckling load = 0.2023E+02

The buckling mode

Node	Translation	Rotation
1	0.0000E+00	0.7273E+00
2	0.1524E+00	0.3884E+00
3	0.1687E+00	-0.2440E+00
4	0.6725E-01	-0.4522E+00
5	0.0000E+00	0.0000E+00

Converged in 12 iterations

**Figure 4.37** Results from first Program 4.6 example



```
nels nprops np_types
4 2 1
```

```
prop(ei,k)
1.0 800.0

etype(not needed)

ell
0.25 0.25 0.25 0.25

nr,(k,nf(:,k),i=1,nr)
2
1 0 1 5 0 1

limit tol
100 1.0e-5
```

**Figure 4.38** Mesh and data for second Program 4.6 example

There are 8 equations and the skyline storage is 23

The buckling load = 0.6003E+02

The buckling mode

Node	Translation	Rotation
1	0.0000E+00	0.5725E+00
2	0.9138E-01	0.1282E-05
3	0.8854E-06	-0.5725E+00
4	-0.9138E-01	-0.1282E-05
5	0.0000E+00	0.5725E+00

Converged in 26 iterations

**Figure 4.39** Results from second Program 4.6 example

indicates a buckling load of 60.03, which compares well with the exact solution of 59.7 (Timoshenko and Gere, 1961). It may also be noted that the fundamental buckling mode is antisymmetric about the centreline, which is to be expected if the foundation stiffness exceeds a threshold value given by  $4\pi^4 EI/L^4$ , which in this case equals about 390. A fuller treatment, including the case of a ‘follower’ force, is given by Smith (1979).

### Program 4.7 Analysis of plates using 4-node rectangular plate elements. Homogeneous material with identical elements. Mesh numbered in $x$ - or $y$ -direction

```

PROGRAM p47
!-----
! Program 4.7 Analysis of plates using 4-node rectangular plate elements.
!           Homogeneous material with identical elements.
!           Mesh numbered in x- or y-direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,      &
nip=16,nlen,nn,nod=4,nodof=4,nprops=2,np_types,nr,nxe,nye
REAL(iwp)::aa,bb,d,d4=4.0_iwp,d12=12.0_iwp,e,one=1.0_iwp,          &
penalty=1.0e20_iwp,th,two=2.0_iwp,v,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g(:, :, :),g_num(:, :, :),kdiag(:,nf(:, :, :), &
no(:,node(:,num(:,sense(:)
REAL(iwp),ALLOCATABLE::bm(:,coord(:, :, dtd(:, :, d2x(:,d2xy(:,d2y(:, &
g_coord(:, :, km(:, :, kv(:,loads(:,points(:, :, prop(:, :, x_coords(:, &
y_coords(:,value(:,weights(:, 
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types,aa,bb,th
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),g_coord(ndim,nn),g_num(nod,nels),g(ndof),bm(3),      &
g_g(ndof,nels),coord(nod,ndim),km(ndof,ndof),dtd(ndof,ndof),d2x(ndof), &
d2y(ndof),d2xy(ndof),num(nod),x_coords(nxe+1),y_coords(nye+1),          &
points(nip,ndim),weights(nip),prop(nprops,np_types),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
DO i=1,nxe+1; x_coords(i)=(i-1)*aa; END DO
DO i=1,nye+1; y_coords(i)=(i-1)*bb; END DO
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq)); kdiag=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    CALL num_to_g(num,nf,g); CALL fkdiag(kdiag,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1
CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)) ) &
WRITE(11,'(2(A,I5))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----element stiffness integration and assembly-----
CALL sample(element,points,weights); kv=zero

```

```

elements_2: DO iel=1,nels
  e=prop(1,etype(iel)); v=prop(2,etype(iel))
  d=e**th**3/(d12*(one-v**2)); g=g_g(:,iel); km=zero
  integration: DO i=1,nip
    CALL fmplat(d2x,d2y,d2xy,points,aa,bb,i)
    DO k=1,ndof
      dtd(k,:)=d4*aa*bb*d*weights(i)*
      (d2x(k)*d2x(:)/(aa**4)+d2y(k)*d2y(:)/(bb**4)+(v*d2x(k)*d2y(:) + &
      v*d2x(:)*d2y(k)+two*(one-v)*d2xy(k)*d2xy(:))/(aa**2*bb**2)) &
      dtd(:,k)=dtd(k,:)
    END DO
    km=km+dtd
  END DO integration
  CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read loads and/or displacements-----
loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)then
  ALLOCATE(node(fixed_freedoms),no(fixed_freedoms),sense(fixed_freedoms),&
  value(fixed_freedoms))
  READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
  DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
  kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
WRITE(11,'(/A)')" Node Disp Rot-x Rot-y Twist-xy"
DO k=1,nn; WRITE(11,'(I5,6E12.4)')k,loads(nf(:,k)); END DO
!-----recover moments at element centroids-----
nip=1; DEALLOCATE(points); ALLOCATE(points(nip,ndim))
CALL sample(element,points,weights)
WRITE(11,'(/A)'") Element x-moment y-moment xy-moment")
DO iel=1,nels
  g=g_g(:,iel)
  mom: DO i=1,nip
    CALL fmplat(d2x,d2y,d2xy,points,aa,bb,i); bm=zero
    DO k=1,ndof
      bm(1)=bm(1)+d4*d*(d2x(k)/aa/aa+v*d2y(k)/bb/bb)*loads(g(k))
      bm(2)=bm(2)+d4*d*(v*d2x(k)/aa/aa+d2y(k)/bb/bb)*loads(g(k))
      bm(3)=bm(3)+d4*d*(one-v)*d2xy(k)/aa/bb*loads(g(k))
    END DO
  END DO mom
  WRITE(11,'(I5,3E12.4)')iel,bm
END DO
STOP
END PROGRAM p47

```

The previous examples have illustrated the principles of finite elements applied to ‘structures’ made up of one-dimensional elements. Solutions to these idealised problems were not usually dependent upon the number of elements, which were chosen conveniently to reflect the positions of the load applications and changes of geometry. This example models a two-dimensional thin plate structure using a genuine finite element approximation. The number of elements used to model the plate is decided by the user, but as the number increases, so the solution should improve. The success of a finite element analysis rests on ‘close enough’ solutions being found using a reasonable number of

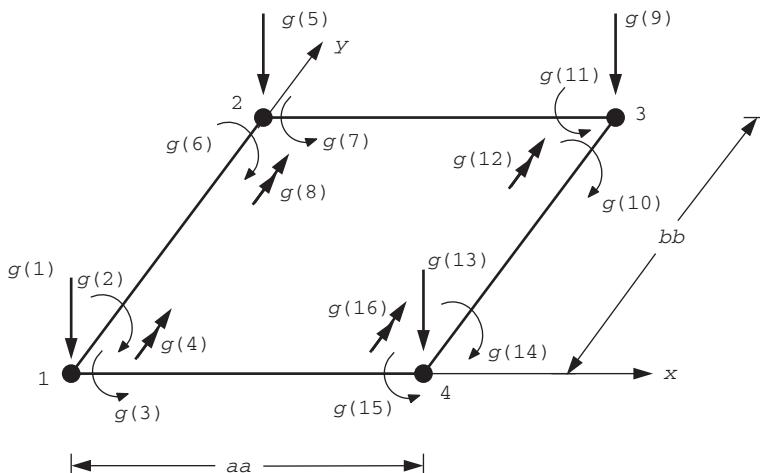
elements. Section 2.14 describes the governing differential equation and the finite element discretisation.

The formulation described here enforces complete compatibility of displacements between elements and equilibrium at the nodes, but there will in general be some loss of equilibrium between the nodes. Figure 4.40 illustrates a typical element and gives the freedom numbering of the  $\mathbf{g}$  vector. It can be seen that there are 16 degrees of freedom per element comprising a vertical translation ( $w$ ), two ordinary rotations ( $\theta_x, \theta_y$ ) and a 'twist' rotation ( $\theta_{xy}$ ) at each node.

The structure of the program is similar to several of the earlier programs in this chapter, except that the element stiffness matrix is calculated using (Gaussian) numerical integration in the  $x$ - and  $y$ -directions. Property data involves Young's modulus and Poisson's ratio, read into the array `prop`, the plate thickness `th` and the rectangular element dimensions `aa` and `bb` (the dimensions of the elements in the  $x$ - and  $y$ -directions respectively). The flexural stiffness of the plate `d` is calculated from the plate thickness and elastic properties. All elements are assumed to be the same size in this program and arranged into a rectangular mesh. This enables the nodal coordinates and 'steering' vector for each element to be generated automatically by a geometry subroutine called `geom_rect`. This subroutine will be used frequently in later chapters of the book, having the ability to generate rectangular meshes for a variety of 2D elements.

In the 'element stiffness integration and assembly' loop, all the derivative arrays mentioned above are delivered for each Gauss point by the library subroutine `fmplat`. Once the Gauss point loop is completed, the element stiffness matrix is held in `km`. This is followed by assembly into a global stiffness matrix, stored as usual as a skyline vector `kv`. Nodal loads (forces, moments) and/or fixed displacements (translations, rotations) are then read and the equilibrium equations solved.

Following calculation of the global displacements, a post-processing phase begins in which the elements are scanned once more. Element nodal displacements are retrieved and the three moments ( $M_x, M_y$  and  $M_{xy}$ ) are computed at the centroid of each element. It should be noted that since `nip=16` was needed for the integration phase (four Gauss

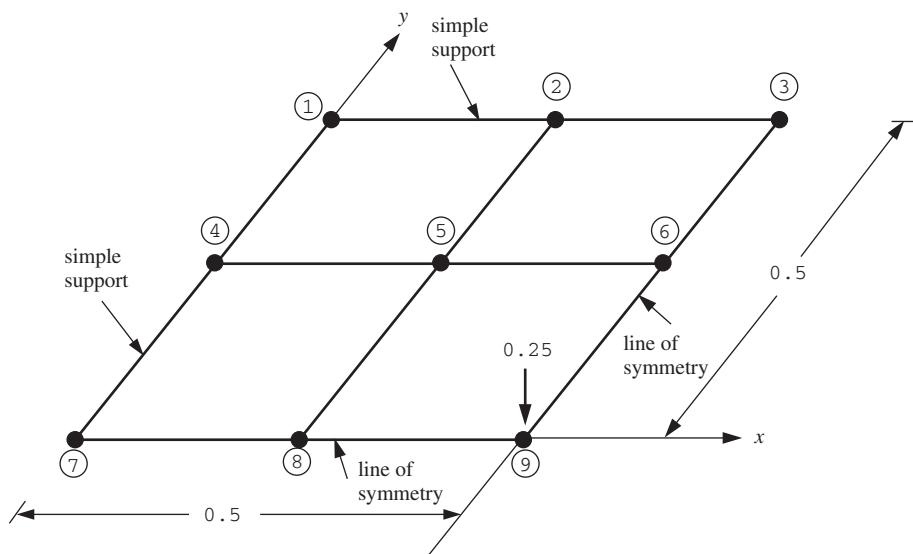


**Figure 4.40** Node and freedom numbering for plate element

points in each of the coordinate directions for exact integration), and `nip=1` is required to find the centroid of each element, it was necessary to reset `nip` to unity and ‘reallocate’ the `points` array.

The example shown in Figure 4.41 illustrates a symmetrical quadrant of a square plate simply supported at its edges and modelled by four square elements. The plate supports a central unit load so one quarter of this value is applied to node 9.

The results in Figure 4.42 show the central deflection of the plate (node 9) to be 0.01147, which can be compared with the ‘exact’ solution of 0.01160 (Timoshenko and Woinowsky-Krieger, 1959). By increasing the number of elements, better approximations to the exact solution are obtained.



```

nx e   ny e   np_types
2      2      1

aa      bb      th
0.25   0.25   1.0

prop(e,v)
10.92  0.3

etype(not needed)

nr,(k,nf(:,k),i=1,nr)
8
1 0 0 0 1   2 0 0 1 1   3 0 0 1 0   4 0 1 0 1
6 1 0 1 0   7 0 1 0 0   8 1 1 0 0   9 1 0 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
1
9  0.25  0.0  0.0  0.0

fixed_freedoms
0

```

**Figure 4.41** Mesh and data for Program 4.7 example

There are 16 equations and the skyline storage is 115

Node	Disp	Rot-x	Rot-y	Twist-xy
1	0.0000E+00	0.0000E+00	0.0000E+00	-0.8743E-01
2	0.0000E+00	0.0000E+00	-0.2021E-01	-0.6814E-01
3	0.0000E+00	0.0000E+00	-0.2956E-01	0.0000E+00
4	0.0000E+00	0.2021E-01	0.0000E+00	-0.6814E-01
5	0.4772E-02	0.1661E-01	-0.1661E-01	-0.6460E-01
6	0.7125E-02	0.0000E+00	-0.2594E-01	0.0000E+00
7	0.0000E+00	0.2956E-01	0.0000E+00	0.0000E+00
8	0.7125E-02	0.2594E-01	0.0000E+00	0.0000E+00
9	0.1147E-01	0.0000E+00	0.0000E+00	0.0000E+00

Element	x-moment	y-moment	xy-moment
1	-0.1193E-01	-0.1193E-01	-0.5555E-01
2	-0.3813E-01	-0.2497E-01	-0.2804E-01
3	-0.2497E-01	-0.3813E-01	-0.2804E-01
4	-0.1211E+00	-0.1211E+00	-0.3347E-01

**Figure 4.42** Results from Program 4.7 example

Program 4.7 is the first program in the book to output a graphics file with the extension \*.msh, which holds a PostScript image of the mesh (see Section 4.1). This file is generated by subroutine mesh, which is one of a suite of graphics subroutines held in the library main. Some of the other subroutines will be described in the next chapter, and are useful for visualising results and debugging data.

## 4.2 Conclusions

It has been shown how sample programs can be built up from the library of subroutines described in Chapter 3. A central feature of the programs has been their brevity. A typical main program has around 100 lines, is comprehensible to the user and is well suited for compilation on a small computer. In subsequent chapters, programs of greater complexity are introduced but the central theme of conciseness is adhered to.

## 4.3 Glossary of Variable Names

### Scalar integers:

fixed_freedoms	number of fixed displacements
i, iel	simple counters
incs	number of load increments
iters	counts plastic iterations
iy	counts load increments
iwp	SELECTED_REAL_KIND(15)
k	simple counter
limit	plastic iteration ceiling
loaded_nodes	number of loaded nodes
ndim	number of dimensions
ndof	number of degrees of freedom per element
nels	number of elements
neq	number of degrees of freedom in the mesh
nip	number of integration points per element
nlen	maximum number of characters in data file basename

nod	number of nodes per element
nodef	number of degrees of freedom per node
nn	number of nodes
np_types	number of different property types
nprops	number of material properties
nr	number of restrained nodes
nxe, nye	number of columns and rows of elements

**Scalar reals:**

aa	element dimension in $x$ -direction
axial	element axial force
bb	element dimension in $y$ -direction
d	plate flexural stiffness
d4, d12	set to 4.0 and 12.0
e	Young's modulus
eval	smallest eigenvalue (buckling load)
one	set to 1.0
penalty	set to $1 \times 10^{20}$
th	plate thickness
tol	plastic or eigenvalue convergence tolerance
total_load	running total of dload ( $= \lambda$ )
two	set to 2.0
v	Poisson's ratio
zero	set to 0.0

**Scalar character:**

argv	holds data file basename
------	--------------------------

**Scalar logical:**

converged	set to .TRUE. if converged achieved
-----------	-------------------------------------

**Dynamic integer arrays:**

etype	element property type vector
g	element steering vector
g_g	global element steering matrix
g_num	global element node numbers matrix
kdiag	diagonal term location vector
nf	nodal freedom matrix
no	fixed freedom numbers vector
node	fixed nodes vector
num	element node numbers vector
sense	sense of freedoms to be fixed vector

**Dynamic real arrays:**

action	element nodal action vector
bddylds	internal correction 'forces' to redistribute moments
bm	bending moments
coord	element nodal coordinates
dload	load increment values

evec	eigenvector (mode shape)
dtd	integration point contribution to km
d2x	second derivatives of shape functions with respect to $\xi$
d2xy	'mixed' second derivatives of shape functions with respect to $\xi$ and $\eta$
d2y	second derivatives of shape functions with respect to $\eta$
eld	element displacement vector
eldtot	keeps a running total of nodal displacements
el1	element lengths vector
gamma	rotation of element about local axis
gm	element geometric matrix
gv	global geometric matrix
g_coord	nodal coordinates for all elements
holdr	holds element 'actions' at convergence
km	element stiffness matrix
kv	global stiffness matrix
loads	global load (displacement) vector
mm	element 'mass' matrix
oldis	nodal displacements from previous iteration
points	holds integration point (local) coordinates
prop	element properties matrix
react	element self-equilibrating 'correction' vector
val	nodal load weightings
value	fixed displacements vector
weights	holds weighting coefficients for numerical integration
x_coords, y_coords	x- and y-coordinates of mesh layout

## 4.4 Exercises

1. A simply supported beam ( $L = 1$ ,  $EI = 1$ ) supports a unit point transverse load ( $Q = 1$ ) at its mid-span. The beam is also subjected to a compressive axial force  $P$  which will reduce the bending stiffness of the beam. Using two ordinary beam elements of equal length, assemble the global matrix equations for this system but do not attempt to solve them. Take full account of symmetries in the expected deformed shape of the beam to reduce the number of equations.

Answer: 
$$\begin{bmatrix} (8 - P/15) & (-24 + P/10) \\ (-24 + P/10) & (96 - 12P/5) \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ -1/2 \end{Bmatrix}$$

2. Derive the mass matrix of a 3-node 1D rod element (one node at each end and one in the middle) of unit length, cross-sectional area and density, given the following shape functions:

$$\begin{aligned} N_1 &= 2(x^2 - 1.5x + 0.5) \\ N_2 &= -4(x^2 - x) \\ N_3 &= 2(x^2 - 0.5x) \end{aligned}$$

Answer: 
$$\frac{1}{30} \begin{bmatrix} 4 & 2 & -1 \\ 2 & 16 & 2 \\ -1 & 2 & 4 \end{bmatrix}$$

3. A cantilever ( $L = 1$ ,  $EI = 1$ ) rests on an elastic foundation of stiffness  $k = 10$ . A transverse point load  $P = 1$  is applied at the cantilever tip. Using a single finite element, estimate the transverse deflection under the load. Check your hand solution using Program 4.3.

Answer: 0.188

4. A propped cantilever is subjected to the loads and displacements indicated in Figure 4.43. Using two finite elements, estimate the internal moment at the centre of the beam. Check your hand solution using Program 4.3.

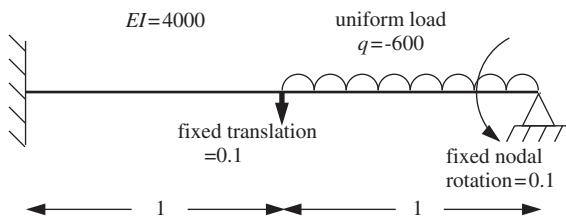


Figure 4.43

Answer:  $M = 1975$

5. Use a single finite element to estimate the lowest buckling load of a column fully fixed at one end and restrained at the other in such a way that it can translate but not rotate as shown in Figure 4.44. Express your solution in terms of  $EI$  and  $L$ . Choose values for  $EI$  and  $L$  and check your hand solution using Program 4.6.

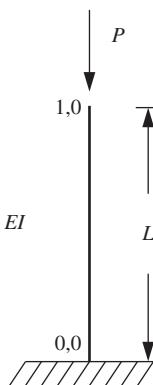


Figure 4.44

Answer:  $P_{crit} = 10EI/L^2$

6. A cantilever of unit length and stiffness supports a unit load at its tip. The governing equation for the transverse deflection  $y$  as a function of  $x$  is given by

$$\frac{d^2y}{dx^2} = 1 - x$$

Estimate the tip deflection using the trial solution

$$\tilde{y} = C(3x^2 - x^3)$$

and Galerkin's weighted residual method.

Answer: 1/3

7. A beam ( $L = 1$ ,  $EI = 1$ ), fully clamped at one end and simply supported at the other, supports a unit point transverse load ( $Q = 1$ ) at its mid-span. The beam is also subjected to a compressive axial force  $P$  which will reduce the bending stiffness of the beam. Using two ordinary beam elements of equal length, assemble the global stiffness equations for this system in matrix form.

Answer:

$$\begin{bmatrix} (192 - 24P/5) & 0 & (24 - P/10) \\ 0 & (16 - 2P/15) & (4 + P/60) \\ (24 - P/10) & (4 + P/60) & (8 - P/15) \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} -1 \\ 0 \\ 0 \end{Bmatrix}$$

8. A simply supported beam of stiffness  $EI$  and length  $2L$  rests on an elastic foundation of stiffness  $k$  and supports a point load  $P$  at its mid-span. By modelling half the beam with a single element, compute the value of  $P = f(EI, k, L)$  that would result in a mid-span negative transverse deflection of 1 unit. Letting  $EI = 1.5$ ,  $k = 10$  and  $L = 2$ , check your solution to the first part of this question by comparing your result with that obtained by Program 4.3.

Answer:  $P = (13kL^2/420 - 6EI/L^2)^2/(kL^3/105 + 4EI/L) - 12EI/L^3 - 13kL/35$

9. Compute the rotation at the middle of the beam shown in Figure 4.45. Check your solution using Program 4.3.

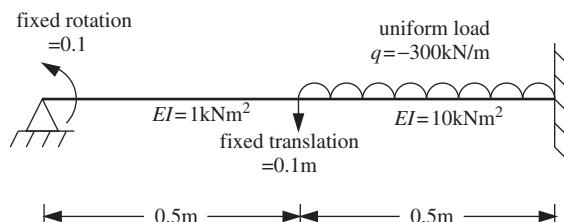


Figure 4.45

Answer:  $\theta = 0.17$

10. A beam of unit length is built in at both ends, has a stiffness  $EI = 1$ , and rests on an elastic foundation as shown in Figure 4.46. A unit load is applied  $1/3$  of the distance from one end. Estimate the value of the foundation stiffness  $k$  such that the deflection under the load is limited to 0.003. Check your solution using trial and error with Program 4.3.

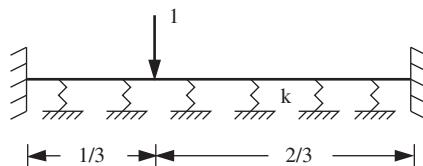


Figure 4.46

Answer:  $k = 123$

11. The continuous beam shown in Figure 4.47 is subjected to an axial load  $P$ . Using beam elements, derive the cubic expression in  $P$  which is given by the buckling loads of the system. Estimate a root of this cubic close to 5. Check your solution using Program 4.6.

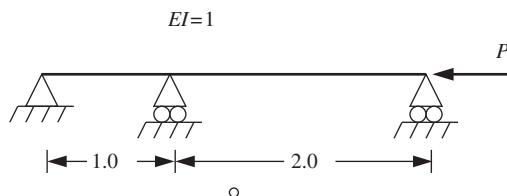


Figure 4.47

Answer:  $P_{crit} = 4.96$

12. Use a simple finite element discretisation to estimate the deflection at point A of the loaded rod shown in Figure 4.48. Check your solution using Program 4.1.

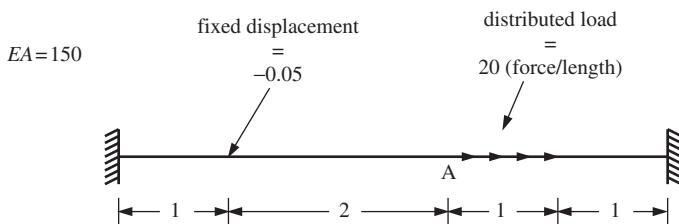


Figure 4.48

Answer: 0.075

13. A beam 20 m long rests on the ground and is to support a uniformly distributed load of  $20 \text{ kN/m}$ . The stiffness of the ground ( $k$ ) and the beam ( $EI$ ) have been estimated at  $10^3 \text{ kN/m}^2$  and  $21 \times 10^4 \text{ kNm}^2$ , respectively. Use two finite elements with simply

supported boundary conditions to estimate the central deflection of the foundation beam. Check your solution using Program 4.3.

Answer: -0.022

14. Use a finite element approach to estimate the lowest buckling load of the beam shown in Figure 4.49 which is supported along half its length by an elastic foundation. Check your solution using Program 4.6.

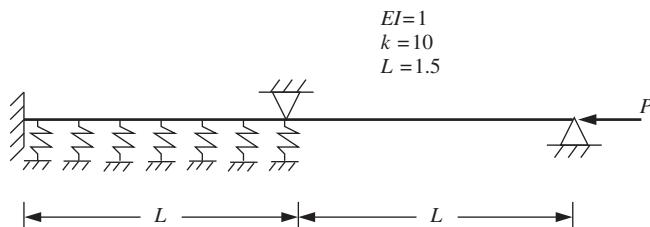


Figure 4.49

Answer:  $P_{crit} = 7.66$

15. Use rod elements to generate the three global stiffness equations due to self-weight of the tower structure shown in Figure 4.50. Let freedom 1 be at the top and freedom 3 be at 2 m above the base. Solve the equations and check your solutions using Program 4.1.

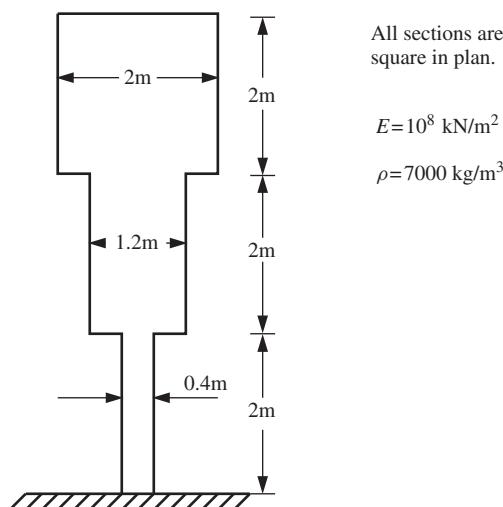


Figure 4.50

$$\text{Answer: } \frac{10^8}{2} \begin{bmatrix} 4 & -4 & 0 \\ -4 & 5.44 & -1.44 \\ 0 & -1.44 & 1.60 \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \\ U_3 \end{Bmatrix} = \begin{Bmatrix} 274.68 \\ 373.56 \\ 109.87 \end{Bmatrix}$$

16. Use beam elements to compute the reaction force  $R_B$  and moment  $M_B$  at the right support of the uniform beam shown in Figure 4.51.

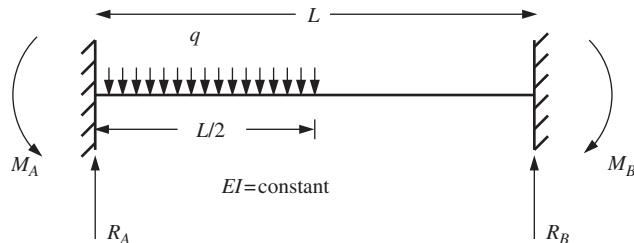


Figure 4.51

Once you have found these values, use global equilibrium equations to compute the corresponding reactions at the left support.

$$\text{Answer: } R_A = \frac{13}{32}qL \quad M_A = \frac{11}{192}qL^2 \\ R_B = \frac{3}{32}qL \quad M_B = -\frac{5}{192}qL^2$$

17. Compute the buckling loads of the beam shown in Figure 4.52. Check the lowest buckling load using Program 4.6.

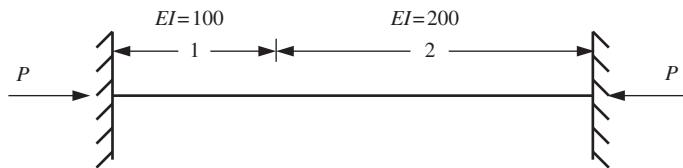


Figure 4.52

$$\text{Answer: } P = 735, P = 2099$$

18. The structure shown in Figure 4.53 consists of three cylindrical sections with the diameters indicated. Using rod finite elements, estimate the deflection of points A and B due to self-weight, and the reaction forces at the top and bottom.

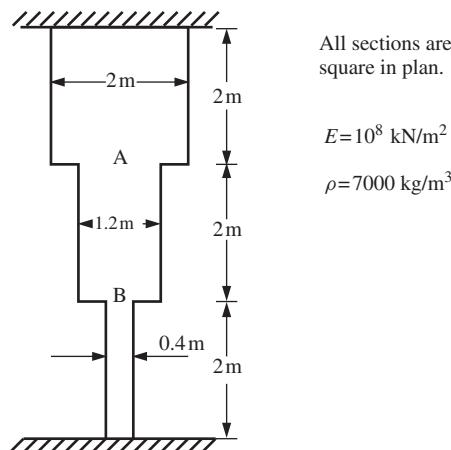


Figure 4.53

Answer:  $\delta_A = 2.280 \times 10^{-6}$  m,  $\delta_B = 3.425 \times 10^{-6}$  m,  
 $R_{top} = 573.9$  kN,  $R_{bot} = 30.2$  kN

19. A framed structure is rigidly attached to a table as shown in Figure 4.54. Use a finite element analysis to compute the force  $Q$  needed to push point A down to the table surface. Check your solution using Program 4.4.

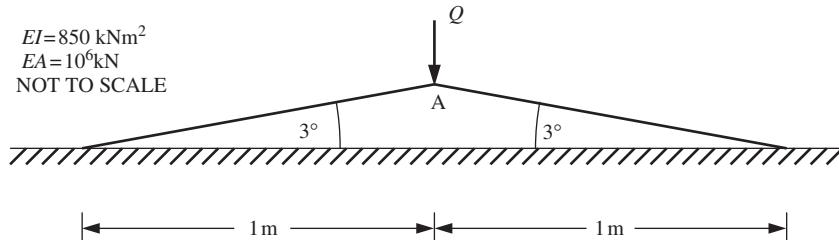


Figure 4.54

Answer:  $Q = 1348.5$  kN

20. A simply supported beam of length  $L$  and stiffness  $EI$  supports a uniformly distributed load  $q$  and rests on an elastic foundation of stiffness  $k$  as shown in Figure 4.55. Use a single beam element to compute the end rotations, and hence estimate the mid-point translation.

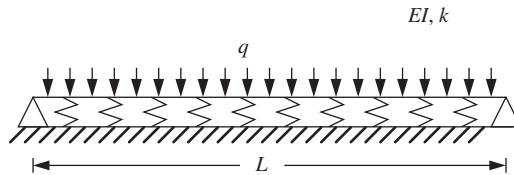


Figure 4.55

Answer:  $\theta = 35qL^3/(840EI + 7kL^4)$ ,  $w_{mid} = 35qL^4/(3360EI + 28kL^4)$

21. Compute the vertical deflection and rotation at the lower end of the system shown in Figure 4.56. Use a single beam–rod element. Check your solution using Program 4.4.

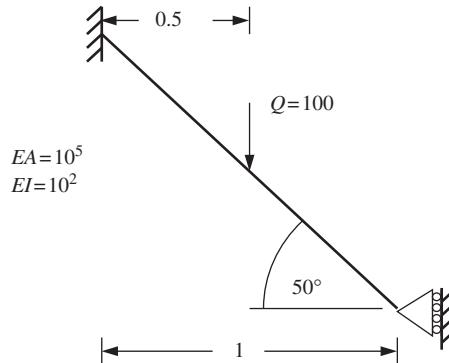


Figure 4.56

Answer:  $U_1 = -0.00112$ ,  $U_2 = 0.04792$

22. The propped cantilever shown in Figure 4.57 has a constant  $E$  and a linearly varying  $I$ . Use two beam elements to estimate the reaction force at B. Check your solution using Program 4.3.

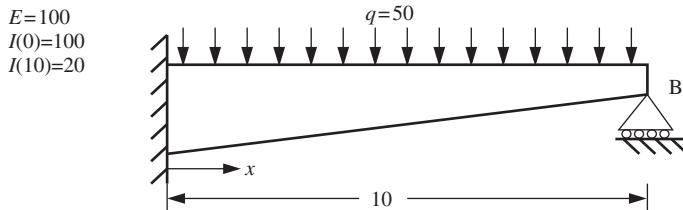


Figure 4.57

Answer:  $R_B = 177.1$  (analytical 175.2)

23. The column shown in Figure 4.58 has been subjected to a gradually increasing axial load  $P$ . The onset of instability was observed when  $P = 102.7$ . Estimate the stiffness  $EI$  of the lower half of the column. Check your solution by trial and error using Program 4.6.

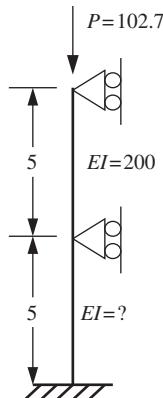


Figure 4.58

Answer:  $EI = 100$

24. A single railway track resting on a ballast subgrade can be approximated as a beam of length  $L$  of stiffness  $EI$  resting on an elastic foundation of stiffness  $k$ . If a single concentrated load  $P$  acts on a rail between the ties which can be assumed to be rigid supports, use a single finite element to estimate the relationship between  $EI$ ,  $k$ ,  $P$  and  $L$  so that the rail deflection can be limited to 5 units. (Hint: Consider the cases of both simply supported and fully clamped end conditions since reality will lie somewhere in between. In the simply supported case you can use one element across the full span and interpolate in the middle. For the fully clamped case you will need to consider just half the problem and account for symmetry.)

Answer: Simply supported,  $P < \frac{320EI}{L^3} + \frac{8kL}{3}$ ; clamped,  $P < \frac{960EI}{L^3} + \frac{13kL}{7}$

25. A laterally loaded pile is to be modelled as a beam on an elastic foundation system as shown in Figure 4.59. Use a single beam element to estimate the lateral deflection of the pile cap under a unit load. (*Hint:* You may assume the base of the pile is clamped.) Check your solution using Program 4.3 with 1 element and 8 elements of equal length.

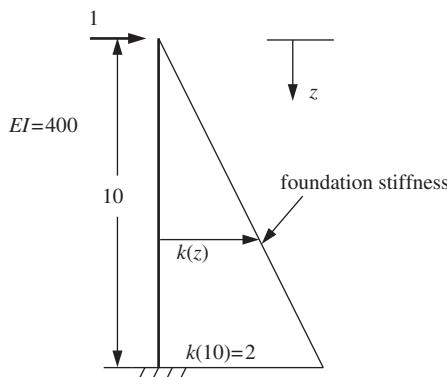


Figure 4.59

- Answer: (1 element) 0.288, (8 elements) 0.488, (exact solution with no constraints on the base) 0.974.
26. Use a single beam element to compute the lowest buckling load of the following cases. You may assume the flexural stiffness and length both equal unity. Check your 1-element solutions using Program 4.6 and show that by mesh refinement your FE solutions converge on the exact values.
- A pin-ended column. Answer: 12, exact solution =  $\pi^2$
  - A column clamped at one end and free at the other. In this case also estimate the ratio of tip rotation to translation when the column buckles. Answer: 2.486, exact solution =  $\pi^2/4$ ; 0.638:1.000
  - A column clamped at both ends. Answer: 40.0, exact solution =  $4\pi^2$
  - A column clamped at both ends with a support preventing deflection at the midpoint. Answer: 120.0, exact solution =  $8.18\pi^2$
27. Buckling of a slender beam of stiffness  $EI$  resting on a uniform elastic foundation of stiffness  $k$  is governed by the equation

$$EI \frac{\partial^4 w}{\partial x^4} + P \frac{\partial^2 w}{\partial x^2} + kw = 0$$

where  $w$  is the transverse deflection of the beam and  $P$  the buckling load.

Using a single finite element, compute two buckling loads for such a beam of length  $L$ , simply supported at its ends. Show that depending on the relationship among  $k$ ,  $EI$  and  $L$ , either of these two loads could be the critical one:

$$\text{if } \theta_1 = -\theta_2, \quad P_{crit} = \frac{12EI}{L^2} + \frac{kL^2}{10}$$

$$\text{if } \theta_1 = \theta_2, \quad P_{crit} = \frac{60EI}{L^2} + \frac{kL^2}{42}$$

Show also that the transition between these two conditions occurs when

$$k = \frac{630EI}{L^4}$$

and check your solutions using Program 4.6.

28. A simply supported beam element of length  $L$ , stiffness  $EI$ , resting on an elastic foundation of stiffness  $k$  supports a uniformly distributed load of  $q$ . Derive a formula for the end rotations using a single finite element and check your result using Program 4.3 after selecting numerical values for  $L$ ,  $EI$  and  $k$ .

Answer:  $\theta = \frac{qL^2/12}{2EI/L + 7kL^3/420}$

29. The simply supported rigid-jointed Vierendeel girder shown in Figure 4.60 has plastic moment values of 90 kNm and 150 kNm in the vertical and horizontal members, respectively. Compute the maximum point vertical load that the beam can support if it is applied at point A or point B.

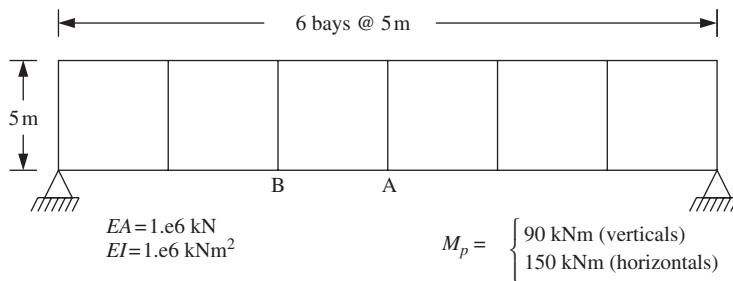


Figure 4.60

Answer: The strength of the beam is *greater* at point A (the centreline) than at point B (Horne, 1971). Use Program 4.5 to give results close to the exact solutions of  $W_{ult_A} = 112$  kN and  $W_{ult_B} = 99$  kN.

## References

- Griffiths DV 1988 An iterative method for plastic analysis of frames. *Comput Struct* **30**(6), 1347–1354.  
 Hetenyi M 1946 *Beams on Elastic Foundations*. University of Michigan Press, Ann Arbor.  
 Horne MR 1971 *Plastic Theory of Structures*. MIT Press, Cambridge, MA.  
 Przemieniecki JS 1968 *Theory of Matrix Structural Analysis*. McGraw-Hill, New York.  
 Smith IM 1979b Discrete element analysis of pile instability. *Int J Numer Anal Methods Geomech* **3**, 205–211.  
 Timoshenko SP and Gere JM 1961 *Theory of Elastic Stability*, 2nd edn. McGraw-Hill, New York.  
 Timoshenko SP and Woinowsky-Krieger S 1959 *Theory of Plates and Shells*. McGraw-Hill, New York.

# 5

# Static Equilibrium of Linear Elastic Solids

## 5.1 Introduction

This chapter describes seven programs which can be used to solve equilibrium problems in small strain solid elasticity. The programs differ only slightly from each other and, following the method adopted in Chapter 4, the first is described in some detail with changes gradually introduced into the later programs. Program 5.1 deals with 2D plane strain or axisymmetric analysis of rectangular regions using any of the 2D elements described in this book. Program 5.2 introduces 3D strain for the special case of non-axisymmetric strain of axisymmetric solids. Program 5.3 introduces conventional 3D analysis of cuboidal meshes offering a choice of hexahedral elements. Program 5.4 is a general program capable of analysing geometrically more complex problems in 2- or 3D including the use of tetrahedral elements. Program 5.5 is a modified version of Program 5.1 accounting for thermoelasticity. The program computes deformations and stresses due to temperature changes. Program 5.6 repeats the analyses described by Program 5.3 using an ‘element-by-element’ pcg technique in which global stiffness assembly is avoided entirely. This procedure lends itself to vectorisation as shown in Program 5.7, which highlights some efficiency issues which arise when programming for a vector computer.

The majority of examples in this chapter consider problems involving a regular (usually rectangular or cuboidal) geometry. This has been done to simplify the presentation and minimise the volume of data required. The simple geometries enable the nodal coordinates and numbers to be generated automatically once the user has provided the element type and preferred numbering direction as data. This is done by geometry subroutines (e.g., `geom_rect` for rectangles and `hexahedron_xz` for cuboids, see Appendix E for geometry subroutine listings). For more complicated geometries, such as are possible using Program 5.4, the geometry subroutines are replaced by READ statements for the nodal coordinates and numbering, and it is left to the user to find some other means of generating this data. Once nodal coordinates, nodal numbering and boundary conditions are known, the next stage in all programs is to determine the element ‘steering vectors’  $\mathbf{g}$ . These are found from `num` and `nf` as in Chapter 4, using the library subroutine `num_to_g`.

To help with debugging and to visualise results, the 2D programs in this chapter include simple graphical subroutines `mesh`, `dismsh` and `vecmsh` which produce PostScript

output files of the undeformed mesh (\*.msh), the deformed mesh (\*.dis) and the nodal displacement vectors (\*.vec), respectively, where ‘\*’ is assigned the basename of the data file. Program 5.6 also introduces the use of subroutines `mesh_ensi` and `dismsh_ensi` for visualisation using ParaView (see Section 1.11).

## 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

```

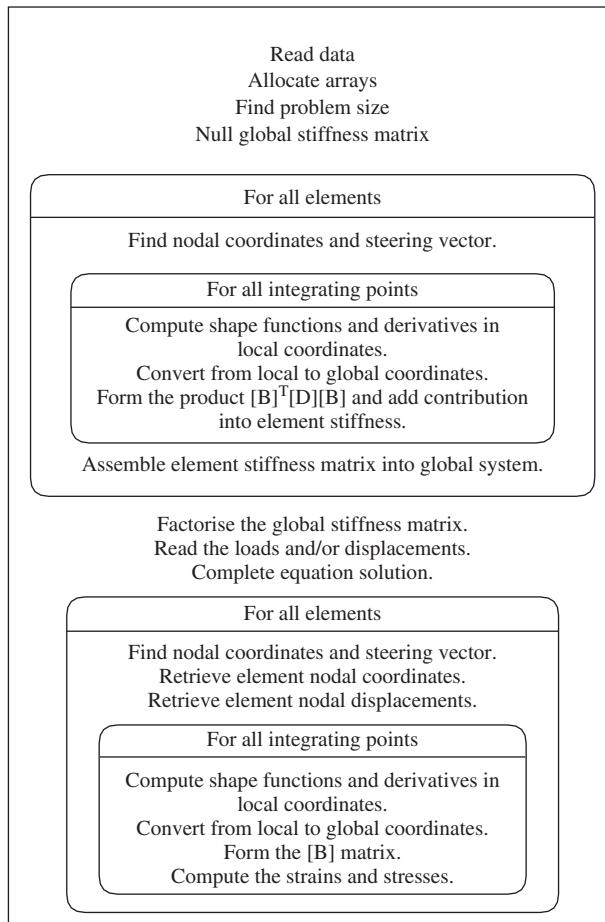
PROGRAM p51
!-----
! Program 5.1 Plane or axisymmetric strain analysis of an elastic solid
!           using 3-, 6-, 10- or 15-node right-angled triangles or
!           4-, 8- or 9-node rectangular quadrilaterals. Mesh numbered
!           in x(r)- or y(z)- direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim=2,ndof,nels,neq,nip,    &
nlen,nn,nod,nodof=2,nprops=2,np_types,nr,nst=3,nxe,nye
REAL(iwp)::det,one=1.0_iwp,penalty=1.0e20_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element,dir,type_2d
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,),g(:,),g_g(:, :,),g_num(:, :,),kdiag(:,),nf(:, :,),  &
no(:, ),node(:, ),num(:, ),sense(:, )
REAL(iwp),ALLOCATABLE::bee(:, :,),coord(:, :,),dee(:, :,),der(:, :,),deriv(:, :,),  &
eld(:, ),fun(:, ),gc(:, ),g_coord(:, :,),jac(:, :,),km(:, :,),kv(:, ),loads(:, ),      &
points(:, :,),prop(:, :,),sigma(:, ),value(:, ),weights(:, ),x_coords(:, ),            &
y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,element,nod,dir,nxe,nye,nip,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); ndof=nod*nodof
IF(type_2d=='axisymmetric')nst=4
ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),fun(nod),&
coord(nod,ndim),jac(ndim,ndim),g_num(nod,nels),der(ndim,nod),          &
deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),eld(ndof),weights(nip),      &
g_g(ndof,nels),prop(nprops,np_types),num(nod),x_coords(nxe+1),           &
y_coords(nye+1),etype(nels),gc(ndim),dee(nst,nst),sigma(nst))
READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(loads(0:neq),kdiag(neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1
CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')
  " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----element stiffness integration and assembly-----

```

```

CALL sample(element,points,weights); kv=zero; gc=one
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    int_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        IF(type_2d=='axisymmetric')THEN
            gc=MATMUL(fun,coord); bee(4,1:nodof-1:2)=fun(:)/gc(1)
        END IF
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)*gc(1)
    END DO int_pts_1
    CALL fsparv(kv,km,g,kdiag)
END DO elements_2
loads=zeros; READ(10,*).loaded_nodes,(k,loads(nf(:,k))),i=1,loaded_nodes)
READ(10,*).fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms), &
              value(fixed_freedoms),no(fixed_freedoms))
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
IF(type_2d=='axisymmetric')THEN
    WRITE(11,'(/A)')" Node r-disp z-disp"; ELSE
    WRITE(11,'(/A)')" Node x-disp y-disp"
END IF
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(nf(:,k)); END DO
!-----recover stresses at nip integrating points-----
nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
CALL sample(element,points,weights)
WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
IF(type_2d=='axisymmetric')THEN
    WRITE(11,'(A,A)')" Element r-coord z-coord", &
    " sig_r sig_z tau_rz sig_t"; ELSE
    WRITE(11,'(A,A)')" Element x-coord y-coord", &
    " sig_x sig_y tau_xy"
END IF
elements_3: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
    int_pts_2: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        IF(type_2d=='axisymmetric')THEN
            gc=MATMUL(fun,coord); bee(4,1:nodof-1:2)=fun(:)/gc(1)
        END IF
        sigma=MATMUL(dee,MATMUL(bee,eld)); WRITE(11,'(I5,6E12.4)')iel,gc,sigma
    END DO int_pts_2
END DO elements_3
CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p51

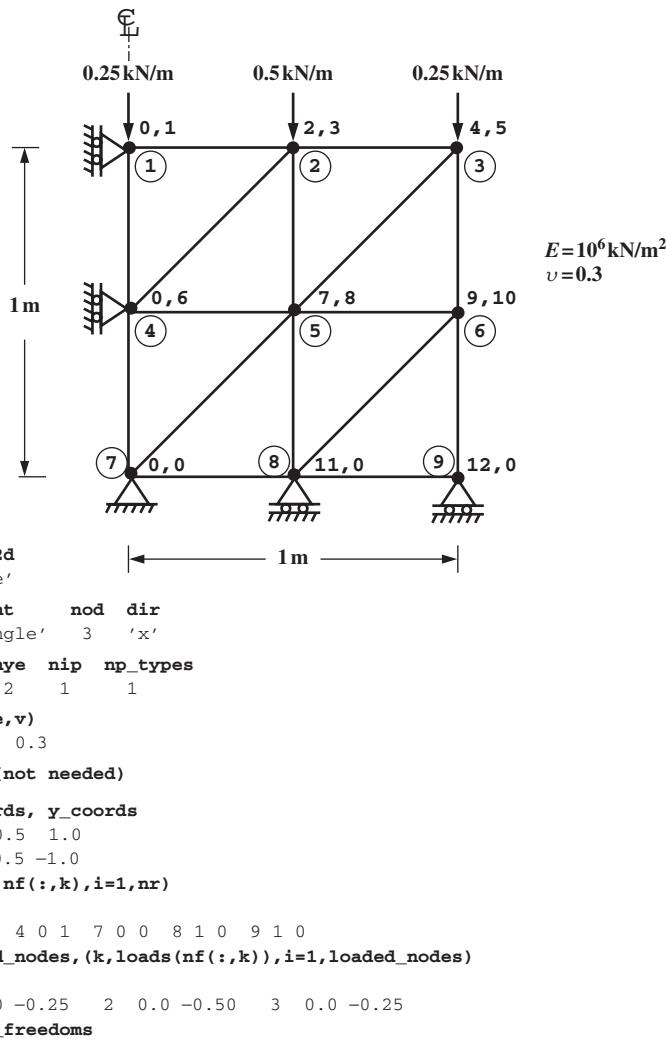
```



**Figure 5.1** Structure chart for all Chapter 5 programs involving assembly

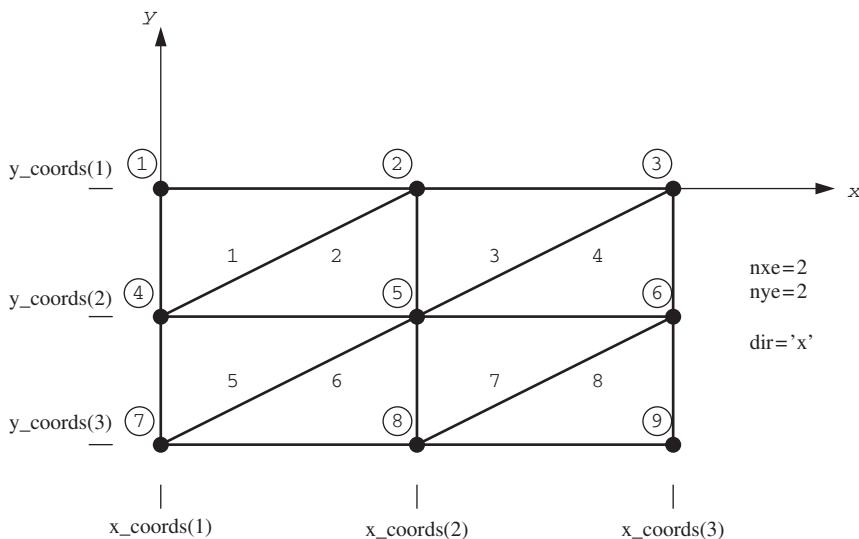
The structure chart in Figure 5.1 illustrates the sequence of calculations for this program. In fact the same chart is essentially valid for all programs in this chapter which use an assembly strategy. Several different 2D elements are available for use by Program 5.1 through the input variables `element` and `nod`. Similarly, the user can select plane or axisymmetric analysis by input to `type_2d` and the numbering direction for nodes and elements by input to `dir`.

The first example for use with Program 5.1 illustrates the use of the simplest 2D element, namely the 3-node (constant strain) triangle. This is not a very good element, and is not used much in practice, except when meshes are automatically adapted to improve accuracy (e.g., Hicks and Mar, 1996). In view of its simplicity, however, the first example in this chapter is devoted to it. As shown in the structure chart, the element stiffness matrices are formed numerically following the procedures described in Chapter 3, in equation (3.13) and Section 3.7.4. For such a simple element, however, only one integrating point (`nip=1`) is required at each element's centroid. Analytical approaches are also possible, as described in Section 3.3.2.

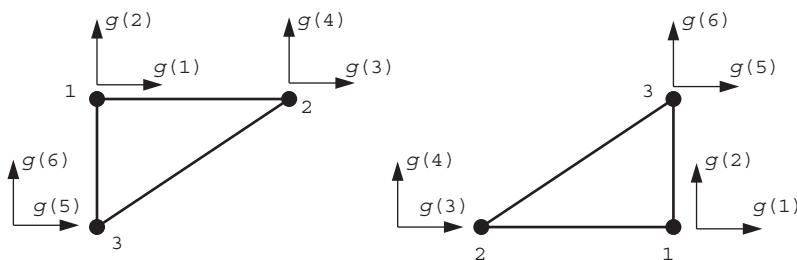


**Figure 5.2** Mesh and data for first Program 5.1 example

Figure 5.2 shows a square block of elastic material of unit side length and unit thickness subjected to an equivalent vertical stress of 1 kN/m<sup>2</sup>. The boundary conditions imply that two planes of symmetry exist and that only one quarter of the problem is being considered. The freedom numbers (not circled) at each node represent possible displacements in the  $x$ - and  $y$ -directions, respectively. Although this information about freedoms is included for completeness, the programs organise the information by nodes and the user need not be aware of freedom numbers at all. Figure 5.3 shows the nodal numbering system adopted for this example and although it does not matter in which direction nodes are numbered for a small problem such as this, the most efficient numbering system for general rectangular shapes will count in the direction with the least nodes. The rectangular



**Figure 5.3** Global node and element numbering for mesh of 3-node triangles



**Figure 5.4** Local node and freedom numbering for different orientations of 3-node triangles

mesh geometry generated by subroutine `geom_rect` assumes that all elements are right-angled, congruent and formed by diagonal lines drawn from the bottom left corner to the top right corner of the surrounding rectangles. Figure 5.3 shows the node and element numbering for this case assuming `dir = 'x'`. Figure 5.4 shows the order of node and freedom numbering at the element level. Node 1 can be any corner, but subsequent corners and freedoms must follow in a clockwise sense. Thus, the top left element (`iel=1`) in Figure 5.2 has a steering vector  $g = [0\ 1\ 2\ 3\ 0\ 6]^T$  and its neighbour (`iel=2`) has a steering vector  $g = [7\ 8\ 0\ 6\ 2\ 3]^T$ .

It is expected that, where necessary, users will replace the geometry subroutine `geom_rect` by more sophisticated versions. It need only be ensured that the coordinates and node numbers are generated consistently.

Referring to Figure 5.2, the first line of data reads the type of strain conditions `type_2d='plane'`, thus a plane-strain analysis is to be performed. The next line reads `element='triangle'`, `nod=3` and `dir='x'`, which indicates that 3-node triangles will be used with node and element numbering in the  $x$ -direction. The next line reads

`nxe=2, nye=2`, which sets the mesh to consist of two columns and two rows of elements, respectively, with the diagonals forming triangles as referred to above, `nip=1` which sets the numerical integration to use one integrating point per element and `np_types` which indicates that there is only one property group in this homogeneous example. As usual, since `np_types` equals 1, the `etype` data is not required. The next line reads the properties which in an elastic analysis consist of Young's modulus and Poisson's ratio, set respectively to  $1 \times 10^6$  kN/m<sup>2</sup> and 0.3. The next two lines read the *x*-coordinates (`x_coords`) and *y*-coordinates (`y_coords`) of the vertical and horizontal lines that make up the mesh. Nodal freedom data concerning boundary restraints is read next, consisting of the number of restrained nodes, `nr=5`, followed by the restrained node number and a binary 'on–off' switch corresponding to the *x*- and *y*-displacement components. For example, `1 0 1` means that at node 1, the *x*-displacement is equal to zero while the *y*-displacement remains free, and `7 0 0` means that node 7 is completely restrained. The final part of the data file refers to loads and fixed displacement data. In this example, a uniform pressure of 1 kN/m<sup>2</sup> is to be applied to the top surface of the block, which in the data file is replaced by equivalent nodal loads. In the case of the 3-node triangle, the total force on each element is simply shared equally between the two nodes (see Appendix A). In this case, `loaded_nodes` is read as 3, representing the nodes at the top of the block, and this is followed by the node number and the *x*- and *y*-components of load to be applied. There are no fixed non-zero displacements in this example, so `fixed_freedoms` is read as zero.

After declaration of arrays whose dimensions are known, the program enters the 'input and initialisation' stage. Data concerning the mesh and its properties are now presented together with the nodal freedom data as given in Figure 5.2. The total number of nodes `nn` and equations in the problem `neq` are provided by subroutine `mesh_size`.

In the section called 'loop the elements to find global arrays sizes', the elements are looped to generate 'global' arrays containing the element node numbers (`g_num`), the element nodal coordinates (`g_coord`) and the element steering vectors (`g_g`). Also within this loop, the array `kdiag` is formed, which holds the addresses of the skyline storage leading diagonal terms (see Figure 3.20). In larger problems, a bandwidth optimiser (e.g., Cuthill and McKee, 1969) will improve efficiency by reordering the nodes. Immediately following this loop, the subroutine `mesh` generates a PostScript file of the mesh held in file `*.msh`, where '\*' is assigned the basename of the data file.

The section called 'element stiffness integration and assembly' is now entered, and begins with a call to subroutine `sample`, which forms the quadrature sampling points and weights. The elements are then looped once more, and the nodal coordinates `coord` and the steering vector `g` for each element are retrieved.

After the stiffness matrix `km` has been nulled, the integration loop is entered. The local coordinates of each integrating point (only 1 in this case) are extracted from `points`, and the derivatives of the shape functions with respect to those coordinates `der` are provided for the 3-node element by the library subroutine `shape_der`. The conversion of these derivatives to the global system `deriv` requires a sequence of subroutine calls described by Equations (3.53)–(3.54). The `bee` matrix is then formed by the subroutine `beemat`.

The next line adjusts the `bee` matrix for axisymmetry if needed and then the contribution from each integration point from (3.57) is scaled by the weighting factor from `weights` and added into the element stiffness matrix `km`. Eventually, the completed `km` is assembled into the global stiffness `kv` using the library subroutine `fsparv` which makes use of `kdiag`, as was used extensively in Chapter 4.

There are 12 equations and the skyline storage is 54

Node	x-disp	y-disp
1	0.0000E+00	-0.9100E-06
2	0.1950E-06	-0.9100E-06
3	0.3900E-06	-0.9100E-06
4	0.0000E+00	-0.4550E-06
5	0.1950E-06	-0.4550E-06
6	0.3900E-06	-0.4550E-06
7	0.0000E+00	0.0000E+00
8	0.1950E-06	0.0000E+00
9	0.3900E-06	0.0000E+00

The integration point (nip= 1) stresses are:

Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.1667E+00	-0.1667E+00	0.0000E+00	-0.1000E+01	-0.8145E-16
2	0.3333E+00	-0.3333E+00	0.3331E-15	-0.1000E+01	-0.1629E-15
3	0.6667E+00	-0.1667E+00	0.1110E-15	-0.1000E+01	0.3665E-15
4	0.8333E+00	-0.3333E+00	0.4441E-15	-0.1000E+01	0.1629E-15
5	0.1667E+00	-0.6667E+00	0.2220E-15	-0.1000E+01	-0.1222E-15
6	0.3333E+00	-0.8333E+00	0.5551E-15	-0.1000E+01	-0.1425E-15
7	0.6667E+00	-0.6667E+00	0.0000E+00	-0.1000E+01	0.1833E-15
8	0.8333E+00	-0.8333E+00	0.2220E-15	-0.1000E+01	-0.4072E-16

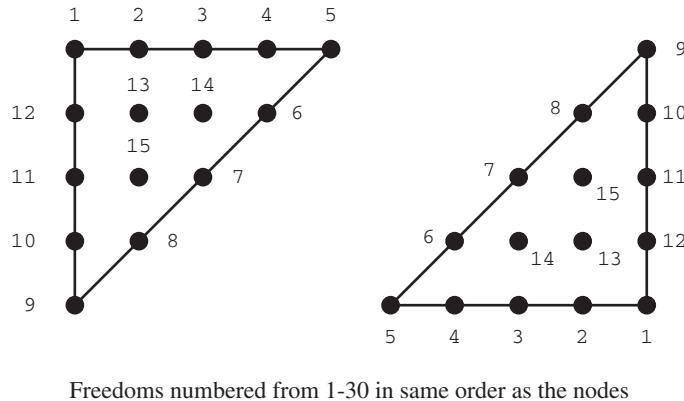
**Figure 5.5** Results from first Program 5.1 example

When all element stiffnesses have been assembled, the program enters the ‘equation solution’ stage. The loads and/or fixed displacements are read, and in the case of fixed displacements, the stiffness matrix  $\mathbf{kv}$  is modified using the ‘stiff spring’ or ‘penalty’ technique (Section 3.6) encountered previously in Chapter 4. The equation solution is in two stages; first, the global matrix  $\mathbf{kv}$  is factorised by subroutine `sparin` and this is followed by the forward and back-substitution stage by subroutine `spabac`. The solution vector holding the nodal displacements (still called `loads`) is printed.

If required, the strains and stresses within the elements can now be computed in the section called ‘recover stresses at nip integrating points’. These could be found anywhere in the elements by computing the  $\mathbf{bee}$  matrix at the required locations, but it is convenient and often more accurate to employ the integrating points that were used in the stiffness formulation. In this example only one integrating point at the element centroid was employed for each element, so it is at this location that strains and stresses will be calculated. Each element is scanned once more and its nodal displacements `eld` retrieved from the global displacement vector `loads`. The  $\mathbf{bee}$  matrix for each integrating point is recalculated and the product of  $\mathbf{bee}$  and `eld` yields the strains from equation (3.58). Multiplication by the stress–strain matrix  $\mathbf{dee}$  gives the stresses  $\sigma$  from equation (3.60) which are printed.

The computed results for the example shown in Figure 5.2 are given in Figure 5.5. For this simple case the results are seen to be ‘exact’. The vertical displacements under the loads (nodes 1, 2 and 3) all equal  $0.91 \times 10^{-6}$  m and the Poisson’s ratio effect has caused horizontal movement at nodes 3, 6 and 9 to equal  $0.39 \times 10^{-6}$  m. The stress components give the expected equilibrium values of  $\sigma_y = -1.0$  and  $\sigma_x = \tau_{xy} = 0$ .

Program 5.1 is able to use both 6-node and 10-node triangular elements, but the next member of the triangular element family to be considered is the 15-node ‘cubic strain’ triangle (see Appendix B). The node numbering system for all triangles involves starting at a corner and progressing clockwise. Internal nodes, if present (e.g., 10- and 15-node triangles) are numbered last. The node numbering at the element level for a 15-node



**Figure 5.6** Local node and freedom numbering for different orientations of 15-node triangles

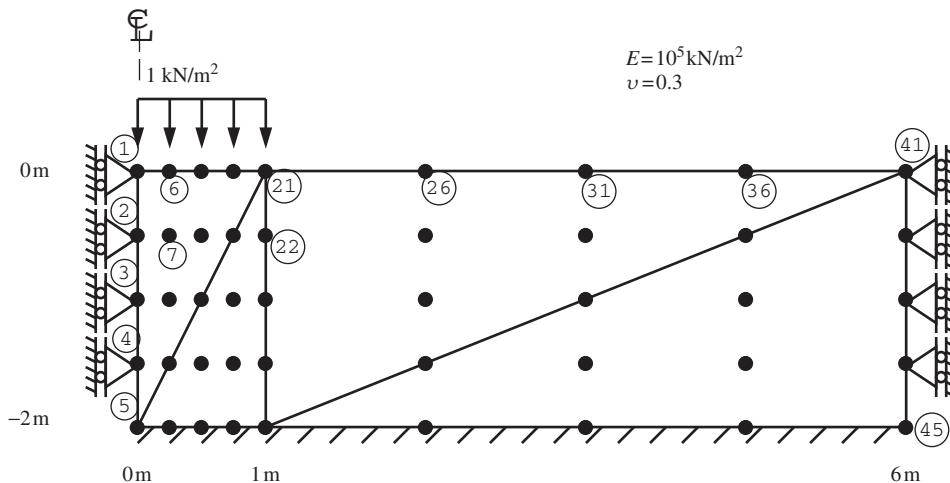
triangle is shown in Figure 5.6. It is seen that the three internal nodes are also numbered in a clockwise sense.

The second example and data in Figure 5.7 show half of a flexible footing resting on a uniform elastic layer supporting a uniform pressure of  $1 \text{ kN/m}^2$ . Because of symmetry, only half of the layer needs to be analysed and the width has been arbitrarily terminated at a roller boundary at six times the load width from the centreline. The data indicate that a 15-node triangle is to be used in a plane-strain analysis, with node and element numbering in the  $y$ -direction. The relatively high order of the interpolating polynomials associated with this element suggests that fewer elements would be required for a typical boundary value problem than if working with a lower-order element. The mesh shown in Figure 5.7 consists of two columns of elements ( $\text{nxe}=2$ ) and one row of elements ( $\text{nye}=1$ ). The recommended number of integrating points for this element in plane strain is  $\text{nip}=12$ . The data follow a similar pattern to the previous example. In this case, the equivalent nodal loads for a 15-node triangle are not intuitive, and Appendix A gives the required values to reproduce a unit pressure under the ‘footing’.

The computed results for this example, given in Figure 5.8, indicate a centreline displacement of  $-0.1591 \times 10^{-4} \text{ m}$ . This is in good agreement with the solution of  $-0.153 \times 10^{-4} \text{ m}$  given by Poulos and Davis (1974). In order to minimise the volume of output, Program 5.1 always computes and prints stresses at element centroids. This is easily achieved in the main program by redefining  $\text{nip}=1$ , followed by a reallocation of the points and weights arrays (having first been ‘deallocated’). Users are of course free to remove these lines, and print the stresses at other locations if required.

The third example demonstrates the 4-node ‘linear strain’ quadrilateral. Figure 5.9 shows a typical mesh of elements, together with the node and element numbering in the case of numbering in the  $y$ -direction ( $\text{dir}='y'$ ). Figure 5.10 gives the node numbering system adopted for the 4-node quadrilateral and also the order in which the recommended number of integrating points  $\text{nip}=4$  are visited. Consistent with triangular elements, nodal numbering always starts at a corner and proceeds clockwise.

Figure 5.11 shows the mesh and data for a rigid strip footing resting on a uniform elastic layer. In this case the footing is given a fixed displacement in the  $y$ -direction of  $-1 \times 10^{-5} \text{ m}$  at nodes 1 and 4 into the layer. Since there are no applied loads,



```

type_2d
'plane'

element nod dir
'triangle' 15 'y'

nxe nye nip np_types
2 1 12 1

prop(e,v)
1.0e5 0.2

etype(not needed)

x_coords, y_coords
0.0 1.0 6.0
0.0 -2.0

nr, (k,nf(:,k),i=1,nr)
17
1 0 1 2 0 1 3 0 1 4 0 1 5 0 0
10 0 0 15 0 0 20 0 0 25 0 0 30 0 0
35 0 0 40 0 0 41 0 1 42 0 1 43 0 1
44 0 1 45 0 0

loaded_nodes, (k,loads(nf(:,k)),i=1,loaded_nodes)
5
1 0.0 -0.0778 6 0.0 -0.3556 11 0.0 -0.1333
16 0.0 -0.3556 21 0.0 -0.0778

fixed Freedoms
0

```

**Figure 5.7** Mesh and data for second Program 5.1 example

`loaded_nodes` is read as zero. The two fixed displacements are entered by reading `fixed_freedoms` as 2, followed by, for each fixed freedom, the node to be fixed (1 and 4), the sense of the fixity (2) and the magnitude of the fixed displacement ( $-1 \times 10^{-5}$  m).

The computed results in Figure 5.12 confirm the fixed y-displacements at nodes 1 and 4 have the expected value of  $-1 \times 10^{-5}$  m. Node 7 has moved up by  $0.1258 \times 10^{-5}$  m, and the vertical stress at the centroid of the element immediately beneath the load gives

There are 64 equations and the skyline storage is 1050

```

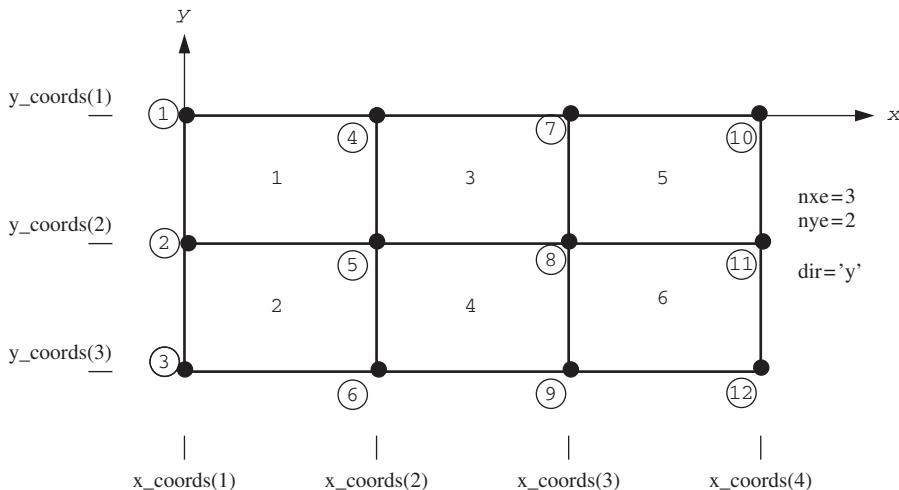
Node   x-disp      y-disp
 1    0.0000E+00 -0.1591E-04
 2    0.0000E+00 -0.1158E-04
 3    0.0000E+00 -0.7226E-05
 4    0.0000E+00 -0.3354E-05
 5    0.0000E+00  0.0000E+00
 6   -0.9321E-06 -0.1559E-04
 7   0.1493E-06 -0.1128E-04
 8   0.4540E-06 -0.7019E-05
 9   0.3347E-06 -0.3255E-05
10   0.0000E+00  0.0000E+00
.
.
.
44   0.0000E+00 -0.1906E-07
45   0.0000E+00  0.0000E+00

```

The integration point (nip= 1) stresses are:

Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.3333E+00	-0.6667E+00	-0.8302E-01	-0.9098E+00	0.7671E+00
2	0.6667E+00	-0.1333E+01	-0.4434E-01	-0.6555E+00	0.1123E+00
3	0.2667E+01	-0.6667E+00	-0.2042E-01	0.3240E-01	-0.1323E-01
4	0.4333E+01	-0.1333E+01	-0.7382E-02	0.1345E-01	-0.3256E-02

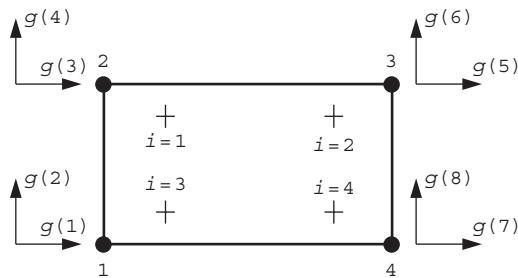
**Figure 5.8** Results from second Program 5.1 example



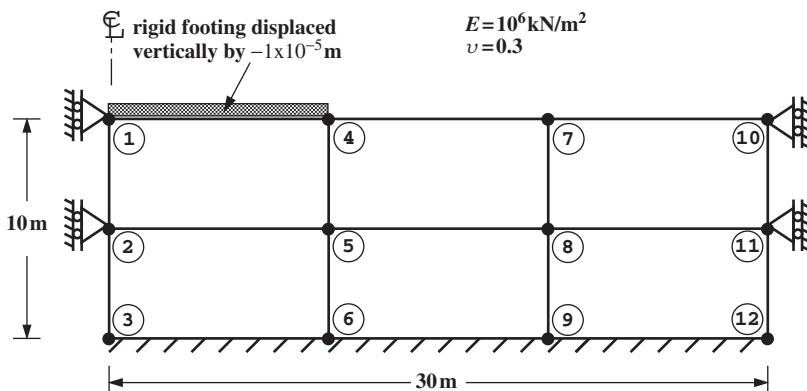
**Figure 5.9** Global node and element numbering for mesh of 4-node quadrilaterals numbered in the y-direction

$\sigma_y = -1.332 \text{ kN/m}^2$ . Comparison with closed-form or other numerical solutions will show that, with such a coarse mesh of these elements, the results can be quite inaccurate. Such discretisation errors are inevitable in finite element work, and it is the user's responsibility to experiment with mesh designs to help discover whether the numerical solution is adequate.

The fourth example, shown in Figure 5.13, illustrates the use of a higher-order element, namely the 8-node 'serendipity' quadrilateral, with nodes and elements numbered



**Figure 5.10** Local node, freedom and Gauss point numbering for the 4-node quadrilateral element ( $nip=4$ )



```

type_2d
'plane'
element          nod   dir
'quadrilateral'    4     'y'
nxe  nxe  nip  np_types
3      2      4      1
prop(e,v)
1.0e6  0.3
etype(not needed)
x_coords, y_coords
0.0  10.0  20.0  30.0
0.0  -5.0  -10.0
nr, (k,nf(:,k),i=1,nr)
8
1 0 1  2 0 1  3 0 0  6 0 0  9 0 0  10 0 1  11 0 1  12 0 0
loaded_nodes
0
fixed_freedoms, (node(i),sense(i),value(i),i=1,fixed_freedoms)
2
1 2  -1.0e-5  4 2  -1.0e-5

```

**Figure 5.11** Mesh and data for third Program 5.1 example

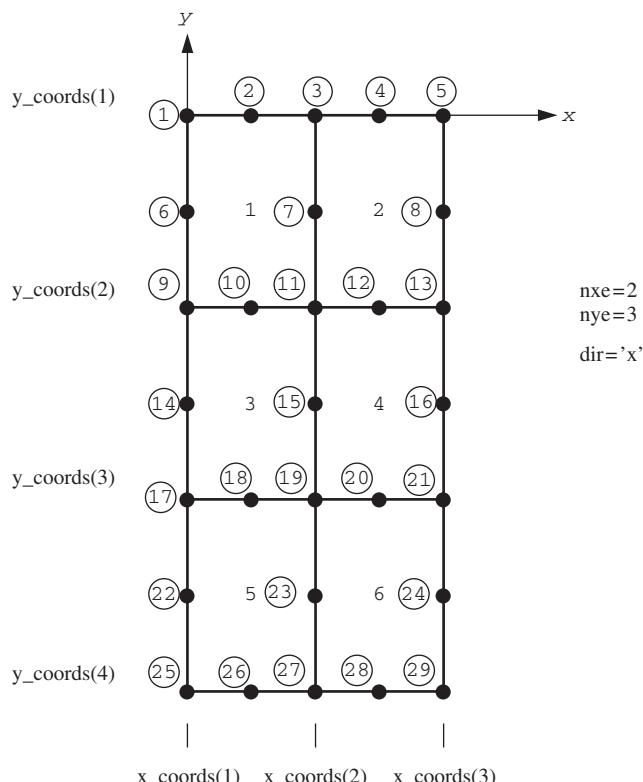
There are 12 equations and the skyline storage is 58

Node	x-disp	y-disp
1	0.0000E+00	-0.1000E-04
2	0.0000E+00	-0.5152E-05
3	0.0000E+00	0.0000E+00
4	0.8101E-07	-0.1000E-04
5	0.1582E-05	-0.4594E-05
6	0.0000E+00	0.0000E+00
7	0.1241E-06	0.1258E-05
8	0.1472E-05	0.1953E-06
9	0.0000E+00	0.0000E+00
10	0.0000E+00	0.2815E-06
11	0.0000E+00	0.3475E-06
12	0.0000E+00	0.0000E+00

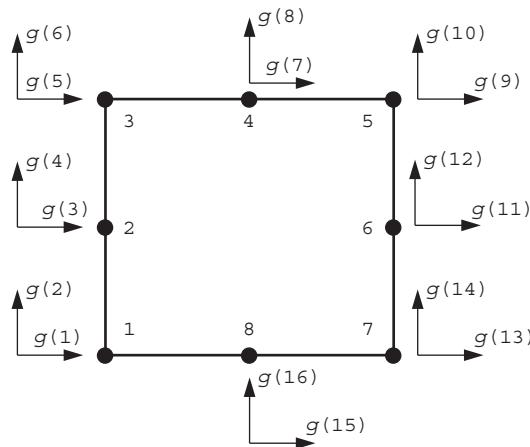
The integration point (nip= 1) stresses are:

Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.5000E+01	-0.2500E+01	-0.4796E+00	-0.1332E+01	-0.4699E-01
2	0.5000E+01	-0.7500E+01	-0.4558E+00	-0.1266E+01	0.7160E-01
3	0.1500E+02	-0.2500E+01	-0.2551E+00	-0.5867E+00	0.1990E+00
4	0.1500E+02	-0.7500E+01	-0.2611E+00	-0.5952E+00	0.2096E+00
5	0.2500E+02	-0.2500E+01	-0.4995E-01	0.8810E-01	-0.6770E-01
6	0.2500E+02	-0.7500E+01	-0.6777E-01	0.3061E-01	0.5955E-01

**Figure 5.12** Results from third Program 5.1 example



**Figure 5.13** Local node and element numbering for mesh of 8-node quadrilaterals numbered in the *x*-direction



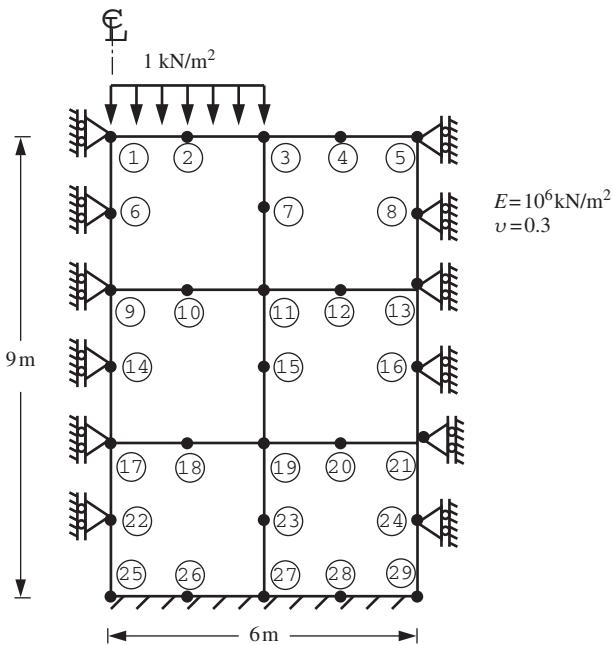
**Figure 5.14** Local node and freedom numbering for the 8-node quadrilateral

in the  $x$ -direction. The local node and freedom numbering for this element as shown in Figure 5.14 indicate, as usual, that node 1 is assigned to a corner and the rest follow in a clockwise sense. The general 8-node quadrilateral element stiffness matrix contains fourth-order polynomial terms and thus requires `nip` to be 9 for ‘exact’ integration. It is often the case, however, that the use of ‘reduced’ integration, by putting `nip` equal to 4, improves the performance of this element. This is found to be particularly true of the plasticity applications described in Chapter 6.

The simple mesh in Figure 5.15 is to be analysed and the consistent nodal loads (Appendix A) necessary to reproduce a uniform stress field of  $1\text{ kN/m}^2$  should be noted in the data. The computed results, given in Figure 5.16, indicate a vertical displacement at node 1 of  $-0.5311 \times 10^{-5}\text{ m}$  and a vertical centroid stress in the element under the load of  $\sigma_y = -0.9003\text{ kN/m}^2$ .

The fifth example and data shown in Figure 5.17 illustrate an axisymmetric foundation analysis (`type_2d = 'axisymmetric'`) as opposed to the plane-strain analyses used in the previous examples. The mesh, while still ‘rectangular’, involves 4-node quadrilateral elements of variable size. Node and element numbering is in the ‘depth’ or ‘ $z$ ’ direction. The mesh size data `nxe` and `nye`, in an axisymmetric context, should be interpreted as the number of ‘columns’ in the radial direction and the number of rows in the depth direction, respectively. Axisymmetric integration is never ‘exact’ using conventional Gaussian quadrature in elastic analysis, due to the  $1/r$  terms that appear in the integrand of the element stiffness matrix [see equations (2.80) and (3.66)]. This example uses `nip=9`, but slightly different results can be expected as `nip` is increased. This example introduces variable properties in which  $E$  and  $\nu$  are allowed to assume different values in each horizontal layer of elements. In this case there are two property groups, so `np_types` is read as 2, and two lots of properties are read into the array `prop`. Since `np_types` is greater than 1, `etype` data is needed and takes the form of integers 1 or 2 for each element, remembering that the mesh elements are numbered in the  $z$ -direction.

It should be noted that in axisymmetry, four components of strain and stress are required, so the main program sets `nst` to 4, and the appropriate `dee` matrix (2.81) is returned by subroutine `deemat`. Furthermore, axisymmetric conditions require the



```

type_2d
'plane'

element      nod   dir
'quadrilateral'    8     'x'

nxe   nye   nip   np_types
2       3       4       1

prop(e,v)
1.0e6  0.3

etype(not needed)

x_coords, y_coords
0.0  3.0  6.0
0.0 -3.0 -6.0 -9.0

nr, (k,nf(:,k), i=1, nr)
17
1  0  1   6  0  1   9  0  1   14 0  1   17 0  1   22 0  1   25 0  0
26 0  0   27 0  0   28 0  0   5  0  1   8  0  1   13 0  1   16 0  1
21 0  1   24 0  1   29 0  0

loaded_nodes, (k, loads(nf(:,k)), i=1, loaded_nodes)
3
1  0.0 -0.5   2  0.0 -2.0   3  0.0 -0.5

fixed_freedoms
0

```

**Figure 5.15** Mesh and data for fourth Program 5.1 example

There are 36 equations and the skyline storage is 390

Node	x-disp	y-disp
1	0.0000E+00	-0.5311E-05
2	-0.4211E-06	-0.5041E-05
3	-0.7222E-06	-0.3343E-05
4	-0.4211E-06	-0.1644E-05
5	0.0000E+00	-0.1375E-05
6	0.0000E+00	-0.4288E-05
7	0.3774E-06	-0.2786E-05
8	0.0000E+00	-0.1283E-05
9	0.0000E+00	-0.3243E-05
10	0.2708E-06	-0.2873E-05
.	.	.
25	0.0000E+00	0.0000E+00
26	0.0000E+00	0.0000E+00
27	0.0000E+00	0.0000E+00
28	0.0000E+00	0.0000E+00
29	0.0000E+00	0.0000E+00

The integration point (nip= 1) stresses are:

Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.1500E+01	-0.1500E+01	-0.2476E+00	-0.9003E+00	0.1040E+00
2	0.4500E+01	-0.1500E+01	-0.1810E+00	-0.9973E-01	0.1040E+00
3	0.1500E+01	-0.4500E+01	-0.1683E+00	-0.6489E+00	0.8714E-01
4	0.4500E+01	-0.4500E+01	-0.2602E+00	-0.3511E+00	0.8714E-01
5	0.1500E+01	-0.7500E+01	-0.1994E+00	-0.5612E+00	0.2888E-01
6	0.4500E+01	-0.7500E+01	-0.2292E+00	-0.4388E+00	0.2888E-01

**Figure 5.16** Results from fourth Program 5.1 example

bee matrix to have a fourth row (2.80), and integration (2.78) involves the radius of each integrating point held in gc(1). The main program checks whether type\_2d is equal to ‘axisymmetry’ and makes these adjustments as necessary.

The nodal loads imply a uniform stress of 1 kN/m<sup>2</sup> is to be applied to a one radian area of radius 10 m (see Appendix A). The computed results for this problem, including stresses at the element ‘centroids’, are given in Figure 5.18. Thus the centreline z-displacement is computed to be  $-0.3176 \times 10^{-1}$  m and the vertical central stress in the depth direction within element 1 is  $\sigma_z = -1.073$  kN/m<sup>2</sup>.

## Program 5.2 Non-axisymmetric analysis of a rectangular axisymmetric elastic solid using 8-node rectangular quadrilaterals. Mesh numbered in r- or z-direction

```
PROGRAM p52
!-----
! Program 5.2 Non-axisymmetric analysis of an axisymmetric elastic solid
!           using 8-node rectangular quadrilaterals. Mesh numbered in
!           r- or z- direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,iflag,k,loaded_nodes,lth,ndim=2,ndof=24,nels,neq,nip=4,      &
nlen,nod=8,nodof=3,nn,nprops=2,np_types,lr,nre,nst=6,nze
REAL(iwp)::ca,chi,det,one=1.0_iwp,pi,radius,sa,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
```

```

!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:,),g_num(:,:,kdiag(:,:nf(:,:, &
num(:)

REAL(iwp),ALLOCATABLE::bee(:,:,coord(:,:,dee(:,:,der(:,:,deriv(:,:)), &
eld(:,fun(:,gc(:,g_coord(:,:,jac(:,:,km(:,:,kv(:),loads(:), &
points(:,:,prop(:,:,r_coords(:,sigma(:,weights(:,z_coords(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nre,nze,lth,iflag,chi,np_types
CALL mesh_size(element,nod,nels,nn,nre,nze)
ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn), &
dee(nst,nst),coord(nod,ndim),fun(nod),jac(ndim,ndim),eld(ndof), &
weights(nip),der(ndim,nod),deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof), &
sigma(nst),num(nod),g_num(nod,nels),g_g(ndof,nels),gc(ndim), &
r_coords(nre+1),z_coords(nze+1),prop(nprops,np_types),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
READ(10,*)r_coords,z_coords
nf=1; READ(10,*)nr,(k,nf(:,k),j=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq))
pi=ACOS(-one); chi=chi*pi/180.0_iwp; ca=COS(chi); sa=SIN(chi); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,r_coords,z_coords,coord,num,'r')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1
CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))') &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----element stiffness integration and assembly-----
CALL sample(element,points,weights); kv=zeros
elements_2: DO iel=1,nels
    CALL deamat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zeros
    gauss_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der)
        CALL bmat_nonaxi(bee,radius,coord,deriv,fun,iflag,lth)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)*radius
    END DO gauss_pts_1
    CALL fsparv(kv,km,g,kdiag)
END DO elements_2
loads=zeros; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
WRITE(11,'(/A)')" Node r-disp z-disp t-disp"
DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----recover stresses at nip integrating points-----
nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
CALL sample(element,points,weights)
WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
WRITE(11,'(A,A)')" Element r-coord z-coord", &
    " sig_r sig_z sig_t" &
WRITE(11,'(A,A)')" ", &
    " tau_rz tau_zt tau_tr" &

```

```

elements_3: DO iel=1,nels
  CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))), num=g_num(:,iel)
  coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
  int_pts_2: DO i=1,nip
    CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
    gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
    deriv=MATMUL(jac,der)
    CALL bmat_nonaxi(bee,radius,coord,deriv,fun,iflag,lth)
    bee(1:4,:)=bee(1:4,:)*ca; bee(5:6,:)=bee(5:6,:)*sa
    sigma=MATMUL(dee,MATMUL(bee,eld))
    WRITE(11,'(I5,5X,5E12.4/34X,3E12.4)') iel,gc,sigma(:3),sigma(4:6)
  END DO int_pts_2
END DO elements_3
CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p52

```

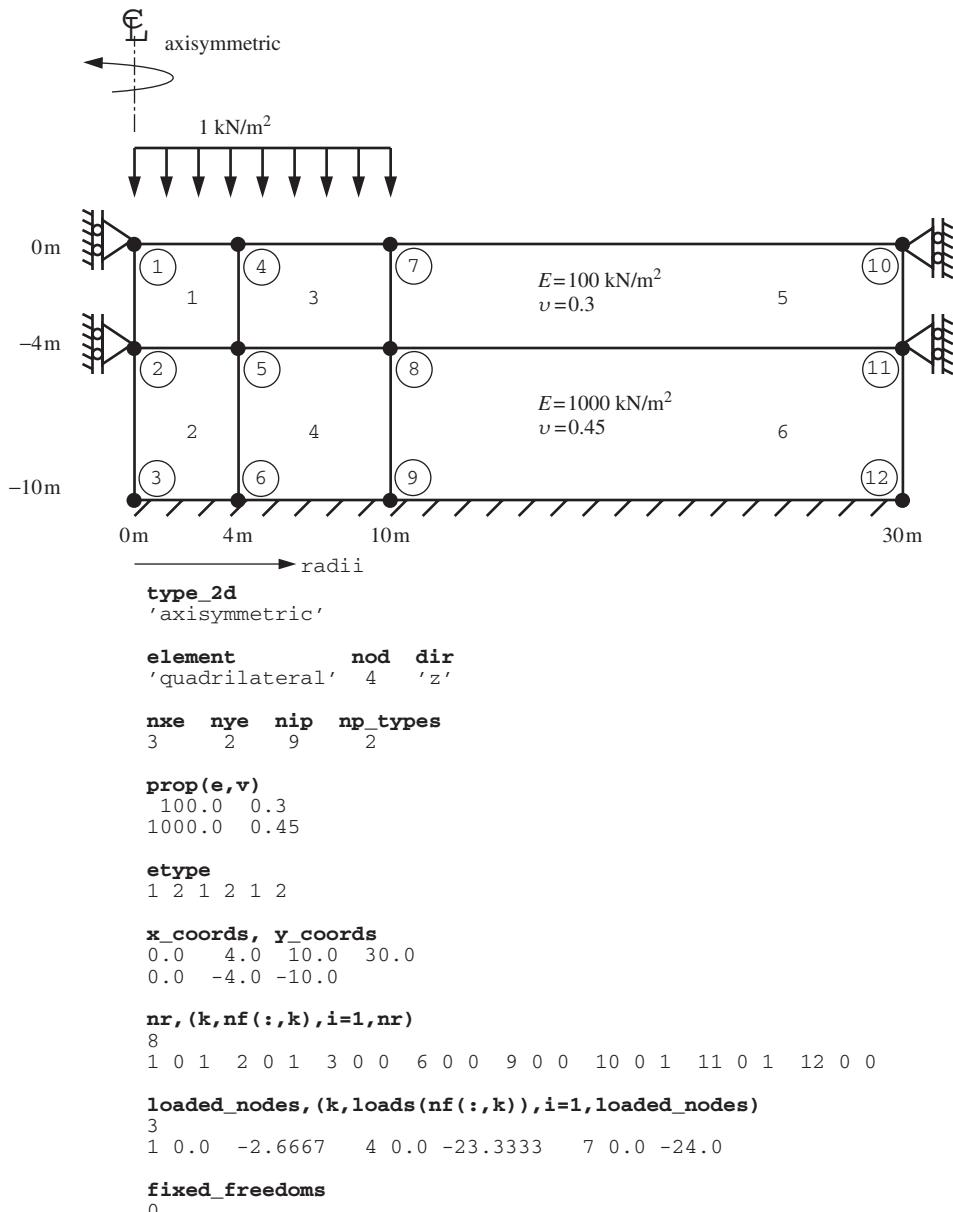
This program allows analysis of axisymmetric solids subjected to non-axisymmetric loads. Variations in displacements, and hence strains and stresses, tangentially are described by Fourier series (e.g., Wilson, 1965; Zienkiewicz *et al.*, 2005). Although the analysis is genuinely three-dimensional, with three degrees of freedom at each node, it is only necessary to discretise the problem in a radial plane. The integration in radial planes is performed using Gaussian quadrature in the usual way. Orthogonality relationships between typical terms in the tangential direction enable the integrals in the third direction to be stated explicitly. The problem therefore takes on the appearance of a two-dimensional analysis with the obvious benefits in terms of storage requirements. The disadvantages of the method over conventional 3D finite element analysis are that (1) the method is restricted to axisymmetric solids and (2) for complicated loading distributions, several loading harmonic terms may be required, and a global stiffness matrix must be stored for each. Several harmonic terms may be required for elastic–plastic analysis (e.g., Griffiths, 1986), but for most elastic analyses such as the one described here, one harmonic will often be sufficient.

It is important to realise that the basic stiffness relationships relate *amplitudes* of load to *amplitudes* of displacement. Once the amplitudes of a displacement are known, the actual displacement at a particular circumferential location is easily found.

For simplicity, consider only the components of nodal load which are symmetric about the  $\theta = 0$  axis of the axisymmetric body. In this case a general loading distribution may be given by

$$\begin{aligned}
 R &= \frac{1}{2} \bar{R}^0 + \bar{R}^1 \cos \theta + \bar{R}^2 \cos 2\theta + \dots \\
 Z &= \frac{1}{2} \bar{Z}^0 + \bar{Z}^1 \cos \theta + \bar{Z}^2 \cos 2\theta + \dots \\
 T &= \bar{T}^1 \sin \theta + \bar{T}^2 \sin 2\theta + \dots
 \end{aligned} \tag{5.1}$$

where  $R$ ,  $Z$  and  $T$  represent the load per radian in the radial, depth and tangential directions. The bar terms with their superscripts represent amplitudes of these quantities on the various harmonics.



**Figure 5.17** Mesh and data for fifth Program 5.1 example

For antisymmetric loading, symmetrical about the  $\theta = \pi/2$  axis, these expressions become

$$\begin{aligned}
R &= \bar{R}^1 \sin \theta + \bar{R}^2 \sin 2\theta + \dots \\
Z &= \bar{Z}^1 \sin \theta + \bar{Z}^2 \sin 2\theta + \dots \\
T &= \frac{1}{2} \bar{T}^0 + \bar{T}^1 \cos \theta + \bar{T}^2 \cos 2\theta + \dots
\end{aligned} \tag{5.2}$$

There are 12 equations and the skyline storage is 58

Node	r-disp	z-disp
1	0.0000E+00	-0.3176E-01
2	0.0000E+00	-0.3231E-02
3	0.0000E+00	0.0000E+00
4	0.1395E-02	-0.3990E-01
5	0.1165E-02	-0.2498E-02
6	0.0000E+00	0.0000E+00
7	0.1704E-02	-0.6046E-02
8	0.1330E-02	-0.4421E-03
9	0.0000E+00	0.0000E+00
10	0.0000E+00	0.2588E-02
11	0.0000E+00	0.3091E-03
12	0.0000E+00	0.0000E+00

The integration point (nip= 1) stresses are:

Element	r-coord	z-coord	sig_r	sig_z	tau_rz	sig_t
1	0.2000E+01	-0.2000E+01	-0.4140E+00	-0.1073E+01	-0.3452E-01	-0.4140E+00
2	0.2000E+01	-0.7000E+01	-0.4776E+00	-0.9072E+00	0.6508E-01	-0.4776E+00
3	0.7000E+01	-0.2000E+01	-0.2933E+00	-0.7099E+00	0.1180E+00	-0.2810E+00
4	0.7000E+01	-0.7000E+01	-0.4316E+00	-0.6101E+00	0.1308E+00	-0.3796E+00
5	0.2000E+02	-0.2000E+01	-0.3200E-01	-0.5814E-01	0.1082E-01	-0.2325E-01
6	0.2000E+02	-0.7000E+01	-0.1090E+00	-0.9367E-01	0.4470E-01	-0.7455E-01

**Figure 5.18** Results from fifth Program 5.1 example

Corresponding to these quantities are amplitudes of displacement in the radial, depth and tangential directions. Since there are now three displacements per node, there are six strains at any point taken in the order

$$\text{eps} = [\epsilon_r \epsilon_z \epsilon_\theta \gamma_{rz} \gamma_{z\theta} \gamma_{\theta r}]^T \quad (5.3)$$

and six corresponding stresses, thus the  $6 \times 6$  stress-strain matrix  $\text{dee}$  (2.88) is formed by the subroutine `deemat` as usual. Using the notation of equation (2.83), the  $[\mathbf{A}]$  matrix now becomes

$$[\mathbf{A}] = \begin{bmatrix} \frac{\partial}{\partial r} & 0 & 0 \\ 0 & \frac{\partial}{\partial z} & 0 \\ \frac{1}{r} & 0 & \frac{1}{r} \frac{\partial}{\partial \theta} \\ \frac{\partial}{\partial z} & \frac{\partial}{\partial r} & 0 \\ 0 & \frac{1}{r} \frac{\partial}{\partial \theta} & \frac{\partial}{\partial z} \\ \frac{1}{r} \frac{\partial}{\partial \theta} & 0 & \frac{\partial}{\partial r} - \frac{1}{r} \end{bmatrix} \quad (5.4)$$

For each harmonic  $i$ , the strain-displacement relationship provided by the library subroutine `bmat_nonaxi` is of the form,

$$\mathbf{B}^i = [\mathbf{B}_1^i \mathbf{B}_2^i \mathbf{B}_3^i \mathbf{B}_4^i \cdots \mathbf{B}_j^i \cdots \mathbf{B}_{\text{nod}}^i] \quad (5.5)$$

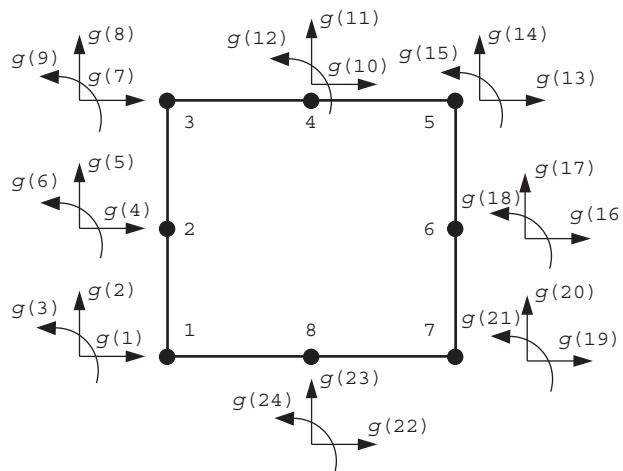
where  $\text{nod}$  is the number of nodes in an element.

A typical submatrix from the above expression for symmetric loading is given by

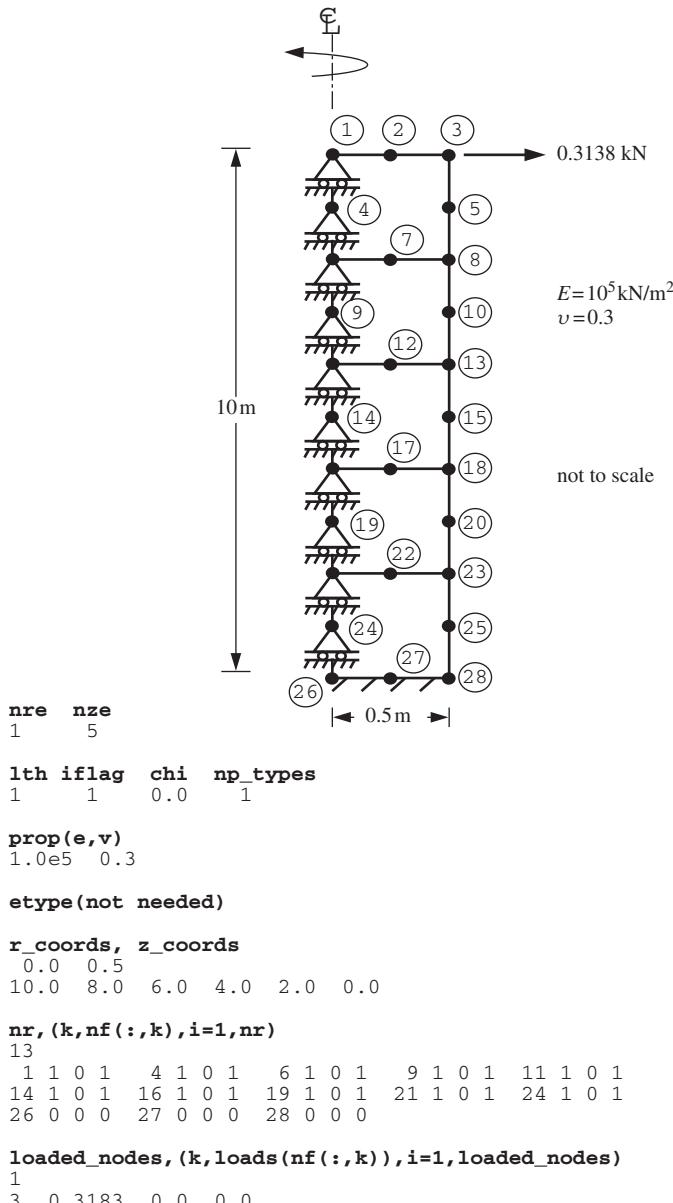
$$[\mathbf{B}]_j^i = \begin{bmatrix} \frac{\partial N_j}{\partial r} \cos i\theta & 0 & 0 \\ 0 & \frac{\partial N_j}{\partial z} \cos i\theta & 0 \\ \frac{N_j}{r} \cos i\theta & 0 & \frac{iN_j}{r} \cos i\theta \\ \frac{\partial N_j}{\partial z} \cos i\theta & \frac{\partial N_j}{\partial r} \cos i\theta & 0 \\ 0 & -\frac{iN_j}{r} \sin i\theta & \frac{\partial N_j}{\partial z} \sin i\theta \\ -\frac{iN_j}{r} \sin i\theta & 0 & \left( \frac{\partial N_j}{\partial r} - \frac{N_j}{r} \right) \sin i\theta \end{bmatrix} \quad (5.6)$$

The equivalent expression for antisymmetry is similar to equation (5.6) but with the sine and cosine terms interchanged and the signs of elements (3,3), (5,2) and (6,1) reversed. Additional INTEGER variables required by this subroutine are iflag and lth. The variable iflag is set to 1 or  $-1$  for symmetry or antisymmetry, respectively, and the variable lth gives the harmonic on which loads are to be applied. An additional variable input to this program is the angle chi (in degrees in the range  $0^\circ$  to  $360^\circ$ ). This is the angle at which stresses are evaluated and printed. Naturally, stresses could be printed at other locations if required. It should be noted that if  $lth=0$  and  $iflag=1$ , the analysis reduces to ordinary axisymmetry as demonstrated by the fifth example with Program 5.1.

The program uses 8-node quadrilateral elements and can be considered a variant of Program 5.1 with nodes and elements numbered in the `dir='r'` direction. Each element has 24 degrees of freedom, as shown in Figure 5.19, and reduced integration (`nip=4`) is assumed.



**Figure 5.19** Local node and freedom numbering for the 8-node quadrilateral (three freedoms per node)



**Figure 5.20** Mesh and data for Program 5.2 example

The example shown in Figure 5.20 represents a homogeneous cylindrical cantilever subjected to a transverse force of 1 kN at its tip. The nature of harmonic loading is such that a radial load amplitude of 1 unit on the first harmonic ( $lth=1$ ) in symmetry ( $iflag=1$ ) results in a net thrust in the  $\theta = 0^\circ$  direction of  $\pi$ . Thus the load amplitude applied at the first freedom of node 3 equals  $1/\pi$ . Along the neutral axis, the nodal freedom data takes account of the fact that there can be no vertical movement along the centreline;

There are 65 equations and the skyline storage is 871

Node	r-disp	z-disp	t-disp
1	0.6755E-01	0.0000E+00	-0.6755E-01
2	0.6755E-01	-0.2528E-02	-0.6755E-01
3	0.6755E-01	-0.5063E-02	-0.6754E-01
4	0.5743E-01	0.0000E+00	-0.5743E-01
5	0.5744E-01	-0.5006E-02	-0.5743E-01
6	0.4753E-01	0.0000E+00	-0.4752E-01
7	0.4753E-01	-0.2426E-02	-0.4752E-01
8	0.4753E-01	-0.4858E-02	-0.4750E-01
9	0.3801E-01	0.0000E+00	-0.3801E-01
10	0.3804E-01	-0.4598E-02	-0.3799E-01

.

.

24	0.9378E-03	0.0000E+00	-0.9620E-03
25	0.1052E-02	-0.9219E-03	-0.8806E-03
26	0.0000E+00	0.0000E+00	0.0000E+00
27	0.0000E+00	0.0000E+00	0.0000E+00
28	0.0000E+00	0.0000E+00	0.0000E+00

The integration point (nip= 1) stresses are:

Element	r-coord	z-coord	sig_r	sig_z	sig_t
			tau_rz	tau_zt	tau_tr
1	0.2500E+00	0.9000E+01	0.6441E+00	-0.5036E+01	-0.4726E+00
			0.8638E-01	0.0000E+00	0.0000E+00
2	0.2500E+00	0.7000E+01	0.2661E+01	-0.1413E+02	0.1144E+01
			0.6800E-01	0.0000E+00	0.0000E+00
3	0.2500E+00	0.5000E+01	0.5441E+01	-0.2274E+02	0.3341E+01
			0.1484E+00	0.0000E+00	0.0000E+00
4	0.2500E+00	0.3000E+01	0.8040E+01	-0.3164E+02	0.5250E+01
			0.3627E+00	0.0000E+00	0.0000E+00
5	0.2500E+00	0.1000E+01	0.1700E+02	-0.3521E+02	0.1580E+02
			0.9873E+00	0.0000E+00	0.0000E+00

**Figure 5.21** Results from Program 5.2 example

hence, these freedoms are restrained. The computed displacements in Figure 5.21 give the end deflection of the cantilever to be  $6.755 \times 10^{-2}$  m, compared with the slender beam value of  $6.791 \times 10^{-2}$  m. If the same load amplitude was applied to the second freedom of node 3, it would correspond to a net moment of 0.5 k Nm. The computed displacement in this case would be  $-5.063 \times 10^{-3}$  m, compared with the slender beam value of  $-5.093 \times 10^{-3}$  m. It should be noted that the current version of Program 5.2 is restricted to load control only.

### Program 5.3 Three-dimensional analysis of a cuboidal elastic solid using 8-, 14- or 20-node brick hexahedra. Mesh numbered in xz-planes then in the y-direction

```

PROGRAM p53
!-----
! Program 5.3 Three-dimensional analysis of an elastic solid using
!           8-, 14- or 20-node brick hexahedra. Mesh numbered in x-y
!           planes then in the z-direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim=3,ndof,nels,neq,nip,   &
nlen,nn,nprops=2,np_types,nod,nodof=3,nr,nst=6,nxe,nye,nze

```

```

REAL(iwp)::det,penalty=1.0e20_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='hexahedron'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:)),g_num(:,:),kdiag(:,nf(:,:, &
no(:,node(:,num(:,sense(:)
REAL(iwp),ALLOCATABLE::bee(:,:,coord(:,:),dee(:,:,der(:,:,deriv(:,:), &
eld(:,fun(:,gc(:,g_coord(:,:),jac(:,:),km(:,:),kv(:,loads(:, &
points(:,:),prop(:,:,sigma(:,value(:,weights(:,x_coords(:, &
y_coords(:,z_coords(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nod,nxe,nye,nze,nip,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye,nze); ndof=nod*nodof
ALLOCATE(nf(ndof,mn),points(nip,ndim),dee(nst,nst),coord(nod,ndim), &
jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),g(ndof),bee(nst,ndof), &
km(ndof,ndof),eld(ndof),sigma(nst),g_g(ndof,nels),g_coord(ndim,nn), &
g_num(nod,nels),weights(nip),num(nod),prop(nprops,np_types), &
x_coords(nxe+1),y_coords(nye+1),z_coords(nze+1),etype(nels),fun(nod), &
gc(ndim))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,z_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(loads(0:neq),kdiag(neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL hexahedron_xz(iel,x_coords,y_coords,z_coords,coord,num)
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))') &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----element stiffness integration and assembly-----
CALL sample(element,points,weights); kv=zero
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); g=g_g(:,iel); coord=TRANSPOSE(g_coord(:,num)); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); jac=MATMUL(der,coord)
        det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
        CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1
    CALL fsparv(kv,km,g,kdiag)
END DO elements_2
loads=zero; READ(10,*)loaded_nodes
IF(loaded_nodes/=0)READ(10,*)(k,loads(nf(:,k)),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms), &
    value(fixed_freedoms),no(fixed_freedoms))
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero

```

```

WRITE(11,'(/A)')" Node      x-disp      y-disp      z-disp"
DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----recover stresses at nip integrating points-----
nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
CALL sample(element,points,weights)
WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
WRITE(11,'(A,,A)')"      Element      x-coord      y-coord      z-coord",
"      sig_x      sig_y      sig_z      tau_xy      tau_yz      tau_zx"
elements_3: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g(:,iel); eld=loads(g)
    gauss_pts_2: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        sigma=MATMUL(dee,MATMUL(bee,eld)); WRITE(11,'(I8,4X,3E12.4)')iel,gc
        WRITE(11,'(6E12.4)')sigma
    END DO gauss_pts_2
    END DO elements_3
STOP
END PROGRAM p53

```

In cases where many Fourier harmonics are required to define a loading pattern, it becomes more efficient and certainly simpler to solve the full three-dimensional problem. Program 5.3 is the 3D counterpart of Program 5.1, and is capable of performing elastic analysis of 3D cuboid meshes consisting of hexahedral brick-shaped elements. The program can incorporate 8-, 14- or 20-node hexahedra and includes a geometry subroutine called hexahedron\_xz for generating meshes in which nodes and elements are counted in  $xz$ -planes moving in the  $y$ -direction as illustrated in Figure 5.22.

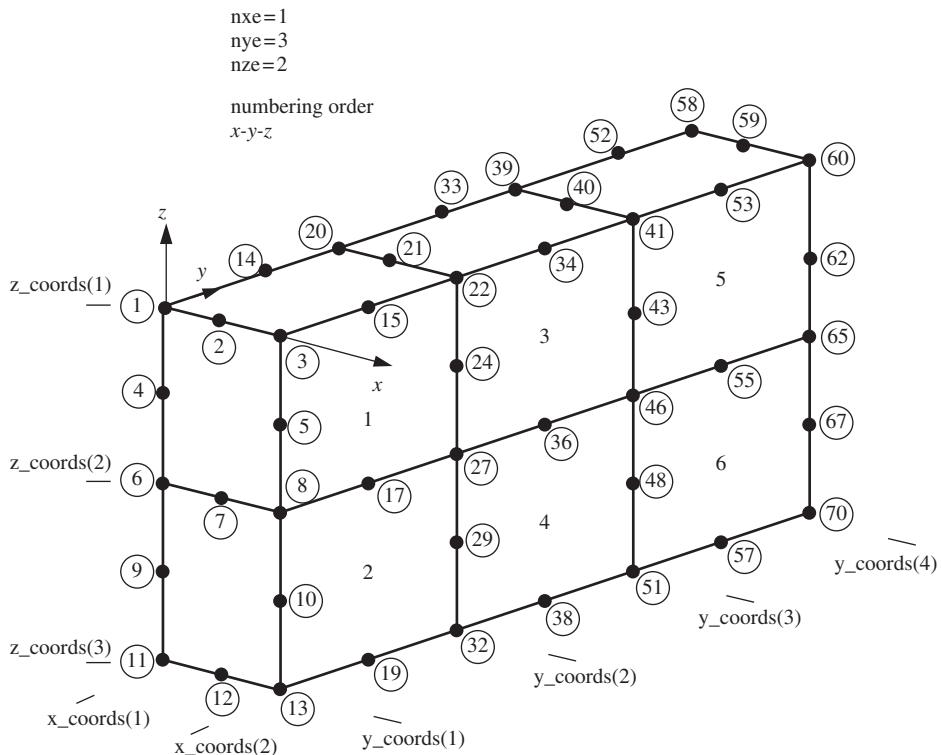
A widely used three-dimensional element, the 20-node hexahedron, is the subject of the example that goes with this program. The element is the 3D analogue of the 8-node quadrilateral in plane problems with the element node numbering indicated in Figure 5.23.

The example and data of Figure 5.24 are for a simple boundary value problem where an elastic block carries a unit uniform load over part of its top surface. The block has layered properties, with the upper and lower halves assigned Young's moduli values of 100 and 50, respectively. Poisson's ratio is fixed at 0.3.

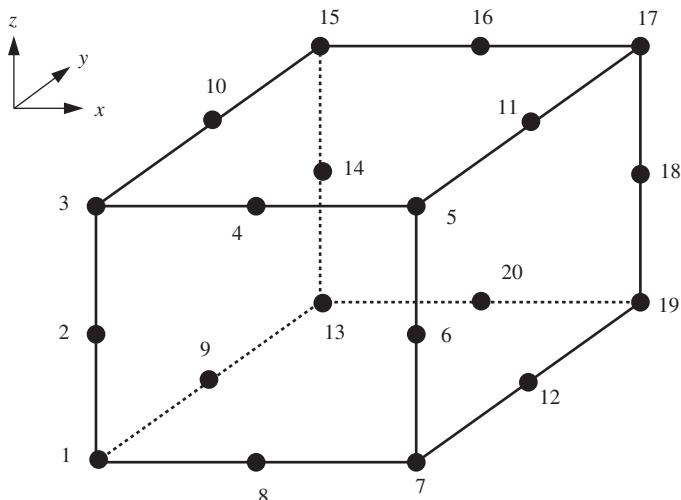
An exact integration scheme for 20-node hexahedral elements would need  $nip=27$ , however 'reduced integration' is recommended for this element, thus  $nip=8$  in the present analysis. The reader can also experiment with Irons's (1971) approximate integration rules, for example,  $nip$  can be set to 6, 14 and 15. The nodal forces to simulate a unit uniform stress field for this element are certainly not intuitive, and involve corner loads which act in the opposite direction to the mid-side loads (Appendix A).

The computed displacements and centroid stresses are given in Figure 5.25. For example, the  $z$ -deflection at the origin of the coordinate system (node 1) is computed as  $-0.2246 \times 10^{-1}$ .

Program 5.3 uses conventional storage and solution strategies, however, storage requirements for 3D analysis rapidly become substantial. It can be noted that even in a simple example such as this, the skyline stiffness vector  $kv$  still requires 4388 locations. For this reason, later programs in this chapter, Programs 5.5 and 5.6, introduce solution methods in which assembly of the global stiffness matrix is avoided entirely.

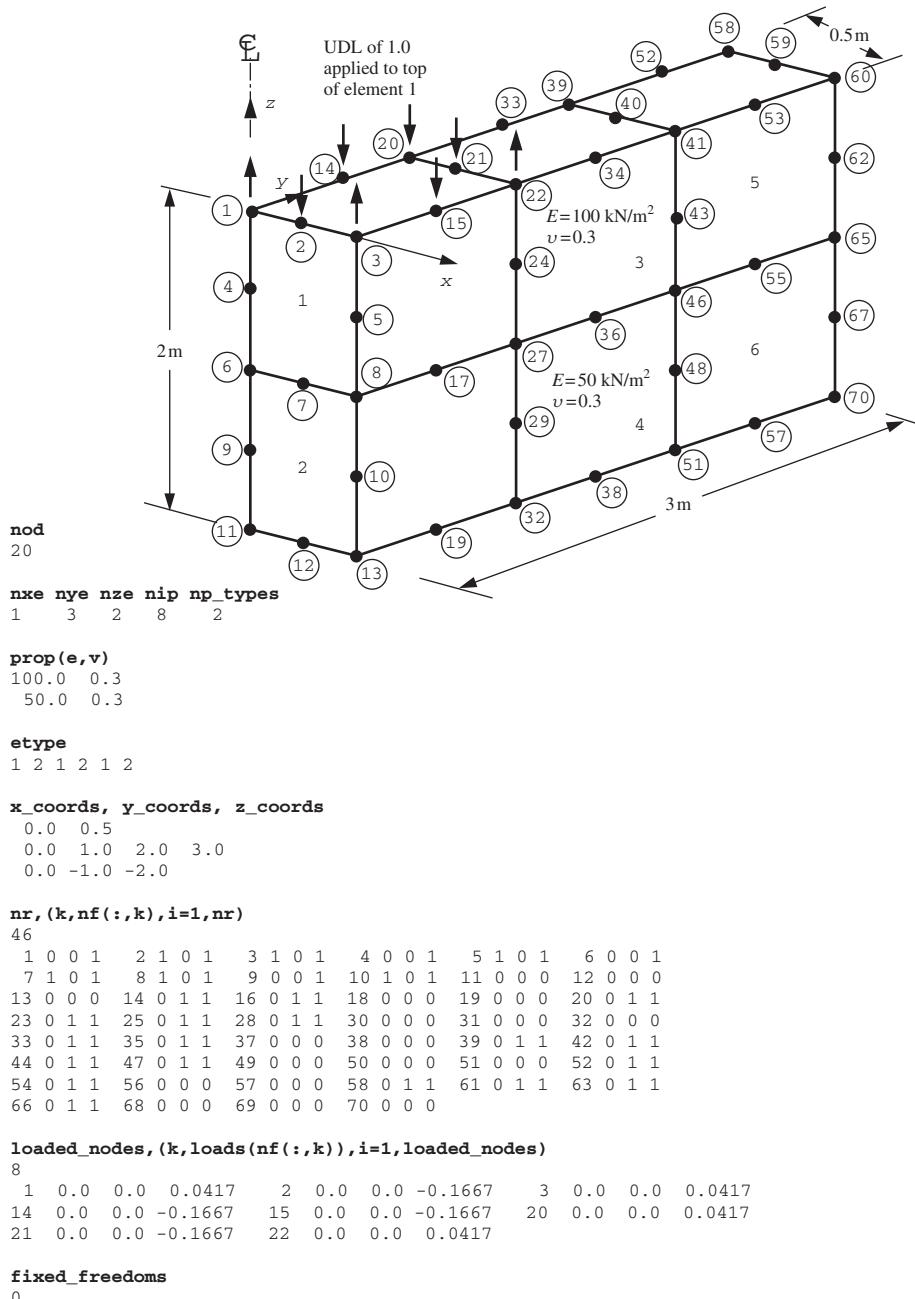


**Figure 5.22** Global node and element numbering for mesh of 20-node hexahedra numbering in the  $x$ -then  $z$ -then  $y$ -direction



Freedoms numbered from 1-60 in same order as the nodes

**Figure 5.23** Local node and freedom numbering for the 20-node hexahedral element

**Figure 5.24** Mesh and data for Program 5.3 example

There are 124 equations and the skyline storage is 4388

Node	x-disp	y-disp	z-disp
1	0.0000E+00	0.0000E+00	-0.2246E-01
2	0.1584E-02	0.0000E+00	-0.2255E-01
3	0.3220E-02	0.0000E+00	-0.2333E-01
4	0.0000E+00	0.0000E+00	-0.1849E-01
5	0.1544E-02	0.0000E+00	-0.1884E-01
6	0.0000E+00	0.0000E+00	-0.1443E-01
7	0.7581E-03	0.0000E+00	-0.1435E-01
8	0.1511E-02	0.0000E+00	-0.1411E-01
9	0.0000E+00	0.0000E+00	-0.6164E-02
10	0.2792E-02	0.0000E+00	-0.6430E-02
.	.	.	.
66	0.0000E+00	0.1572E-02	-0.1028E-03
67	-0.7444E-04	0.1716E-02	-0.5846E-04
68	0.0000E+00	0.0000E+00	0.0000E+00
69	0.0000E+00	0.0000E+00	0.0000E+00
70	0.0000E+00	0.0000E+00	0.0000E+00

The integration point (nip= 1) stresses are:

Element	x-coord	y-coord	z-coord	sig_x	sig_y	sig_z	tau_xy	tau_yz	tau_zx
1	0.2500E+00	0.5000E+00	-0.5000E+00	-0.2672E-01	-0.1647E+00	-0.9088E+00	0.6145E-02	0.9597E-01	0.4352E-02
2	0.2500E+00	0.5000E+00	-0.1500E+01	0.3985E-01	-0.5316E-01	-0.6298E+00	-0.2140E-02	0.7614E-01	0.4169E-02
3	0.2500E+00	0.1500E+01	-0.5000E+00	-0.2482E-01	-0.1260E+00	-0.1052E+00	0.4840E-02	0.9399E-01	-0.2814E-02
4	0.2500E+00	0.1500E+01	-0.1500E+01	0.2477E-01	-0.8240E-01	-0.2822E+00	-0.3179E-02	0.1214E+00	0.2939E-02
5	0.2500E+00	0.2500E+01	-0.5000E+00	0.3767E-02	0.8619E-02	-0.1469E-01	-0.9006E-03	-0.8733E-02	0.8652E-03
6	0.2500E+00	0.2500E+01	-0.1500E+01	0.7407E-02	-0.4390E-01	-0.2831E-01	-0.5639E-03	0.6083E-01	0.5078E-04

Figure 5.25 Results from Program 5.3 example

## Program 5.4 General 2D (plane strain) or 3D analysis of elastic solids. Gravity loading option

```

PROGRAM p54
!-----
! Program 5.4 General two- (plane strain) or three-dimensional analysis
!          of elastic solids (optional gravity loading).
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed Freedoms,i,iel,k,loaded_nodes,ndim,ndof,nels,nip,nlen,&
nn,nod,nodof,nprops=3,np_types,nr,nst
REAL(iwp)::det,penalty=1.0e20_iwp,zero=0.0_iwp
CHARACTER(len=15)::argv,element
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:, :,g_num(:, :,kdiag(:, nf(:, :,&
no(:, node(:, num(:, sense(:)
REAL(iwp),ALLOCATABLE::bee(:, :,coord(:, :,dee(:, :,der(:, :,deriv(:, :,&
eld(:, fun(:, gc(:,gravlo(:, g_coord(:, :,jac(:, :,km(:, :,kv(:, ,&
loads(:, points(:, :,prop(:, :,sigma(:, value(:,weights(:,&
!-----input and initialisation-----
CALL getname(argv,nlen)

```

```

OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)element,nod,nels,nn,nip,nodof,nst,ndim,np_types; ndof=nod*nodof
ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),g_coord(ndim,nn), &
coord(nod,ndim),jac(ndim,ndim),weights(nip),num(nod),g_num(nod,nels), &
der(ndim,nod),deriv(ndim,nod),bee(nst,nodof),km(ndof,ndof),eld(ndof), &
sigma(nst),g(ndof),g_g(ndof,nels),gc(ndim),fun(nod),etype(nels), &
prop(nprops,np_types))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)g_coord; READ(10,*)g_num
IF(ndim==2)CALL mesh(g_coord,g_num,argv,nlen,12)
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formmf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),gravlo(0:neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq))) &
WRITE(11,'(2(A,I5))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----element stiffness integration and assembly-----
CALL sample(element,points,weights); kv=zero; gravlo=zero
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; eld=zero
    int_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(transpose(bee),dee),bee)*det*weights(i)
        eld(nodof:nodof:nodof)=eld(nodof:nodof:nodof)+fun(:)*det*weights(i)
    END DO int_pts_1
    CALL fsparv(kv,km,g,kdiag); gravlo(g)=gravlo(g)-eld*prop(3,etype(iel))
END DO elements_2
loads=zeros; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
loads=loads+gravlo; READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms), &
    value(fixed_freedoms),no(fixed_freedoms))
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
IF(ndim==3)THEN; WRITE(11,'(/A)')" Node x-disp y-disp z-disp"
ELSE; WRITE(11,'(/A)')" Node x-disp y-disp"
END IF
DO k=1,nn; WRITE(11,'(I5,3E12.4)')k,loads(nf(:,k)); END DO
!-----recover stresses at element Gauss-points-----
!nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
!CALL sample(element,points,weights)
WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
IF(ndim==3)THEN
    WRITE(11,'(A,,A)')" Element x-coord y-coord z-coord", &
    " sig_x sig_y sig_z tau_xy tau_yz tau_zx"
ELSE; WRITE(11,'(A,A)')" Element x-coord y-coord", &
    " sig_x sig_y tau_xy"

```

```

END IF
elements_3: DO iel=1,nels
  CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
  coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
  int_pts_2: DO i=1,nip
    CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
    gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    sigma=MATMUL(dee,MATMUL(bee,eld))
    IF(ndim==3)THEN; WRITE(11,'(I8,4X,3E12.4)')iel,gc
      WRITE(11,'(6E12.4)')sigma
    ELSE; WRITE(11,'(I8,2E12.4,5X,3E12.4)')iel,gc,sigma
    END IF
  END DO int_pts_2
END DO elements_3
IF(ndim==2)THEN; CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
  CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
END IF
STOP
END PROGRAM p54

```

Perusal of Programs 5.1 and 5.3 in this chapter will show that they are essentially identical. The shape function and derivative subroutines `shape_fun` and `shape_der`, and the **[B]** and **[D]** subroutines `beemat` and `deemat`, can all generate the appropriate terms once the 2D or 3D element type has been identified through the data. Program 5.4 utilises this identity of programs to create a single general program. Of course, such a program will expect to read the nodal geometry `g_coord` and connectivity `g_num` details from a file which would usually be provided by a mesh-generation pre-processor. The element types available in the library are, for plane strain or axisymmetry:

3-, 6-, 10- and 15-node triangles  
4-, 8- and 9-node quadrilaterals

and for 3D:

4-node tetrahedra  
8-, 14- and 20-node hexahedra

The local numbering of all these elements, which is needed for input to Program 5.4, is given in Appendix B. In addition to the usual two elastic parameters of Young's modulus and Poisson's ratio, this program also allows the option of gravity load generation through the unit weight, which must be read in as a third property (`nprops=3`) for each property group. If gravity loading is not required, the unit weight is read as zero.

The global gravity loading vector (called `gravlo` in the program) for a material with unit weight  $\gamma$  is accumulated from each element by integration of the shape functions **[N]** as follows:

$$\text{gravlo} = \sum_{\text{elements}}^{\text{all}} \gamma \int \int [\mathbf{N}]^T dx dy \quad (5.7)$$

and these calculations are performed in the same part of the program that forms the global stiffness matrix. It may be noted that only those freedoms corresponding to vertical movement are incorporated in the integrals. At the element level, the 1D array `eld` is

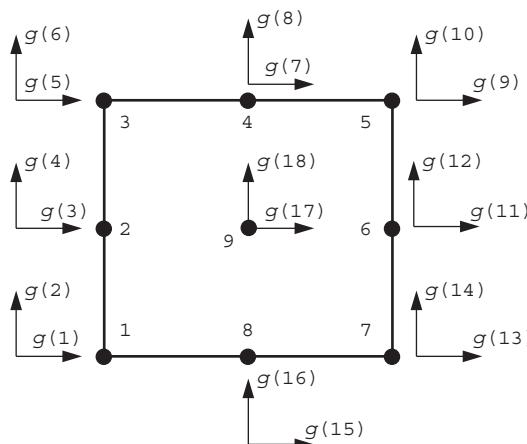
used to gather the contributions from each Gauss point. The global gravity loads vector `gravlo` accumulates `elld` from each element after multiplication by the unit weight  $\gamma$  held in the `prop` array.

The first example uses a 9-node ‘Lagrangian’ element with numbering shown in Figure 5.26. The example problem in plane strain shown in Figure 5.27 is deliberately chosen to allow a comparison with a similar problem previously analysed by Program 5.1 using 8-node elements (Figures 5.15 and 5.16). No gravitational loading has been included in this example, and ‘exact’ integration has been used by setting `nip` equal to 9. This version of Program 5.4 elects to print stresses at all the integrating points used in the stiffness integration, thus the output file prints stresses at nine locations per element. The results given in Figure 5.28 indicate a centreline displacement of  $-0.5299 \times 10^{-5}$  m, and a centroid stress  $\sigma_y$  in the first element (also located at the central node) of  $-0.8766 \text{ kN/m}^2$ .

This could be compared with the centreline displacement of  $-0.5311 \times 10^{-5}$  and centroid stress of  $-0.9003$  from Figure 5.16 using the 8-node element. Users can experiment with the influence of gravity. In this example, if the third property representing the unit weight is set to 1.0, and `loaded_nodes` is set to zero (gravity loading only), the vertical stress  $\sigma_y$  is computed to be identical to the depth of each integrating point.

The second example illustrates the use of the simplest 3D element, namely the 4-node tetrahedron with numbering given in Figure 5.29. This ‘constant-strain’ element is analogous to the 3-node triangle for plane problems described in Program 5.1 and, like the triangle, is not recommended for practical calculations unless adaptive mesh refinement has been implemented. Like the 3-node triangle, this element is exactly integrated using `nip=1`. The example in Figure 5.30 represents a homogeneous cube made up of six tetrahedra. One corner of the cube is fixed and the three adjacent faces are restrained to move only in their own planes. The four nodal forces applied are equivalent to a uniform vertical compressive stress of  $1 \text{ kN/m}^2$  (Appendix A). The computed results given in Figure 5.31 show that the cube compresses uniformly and that the vertical stress  $\sigma_z$  at the centroid is equal to unity, and in equilibrium with the applied loads.

The simplest member of the hexahedral or ‘brick’ element family has eight nodes, situated at the corners, however the element is quite ‘stiff’ in certain deformation modes



**Figure 5.26** Local node and freedom numbering for the 9-node quadrilateral element

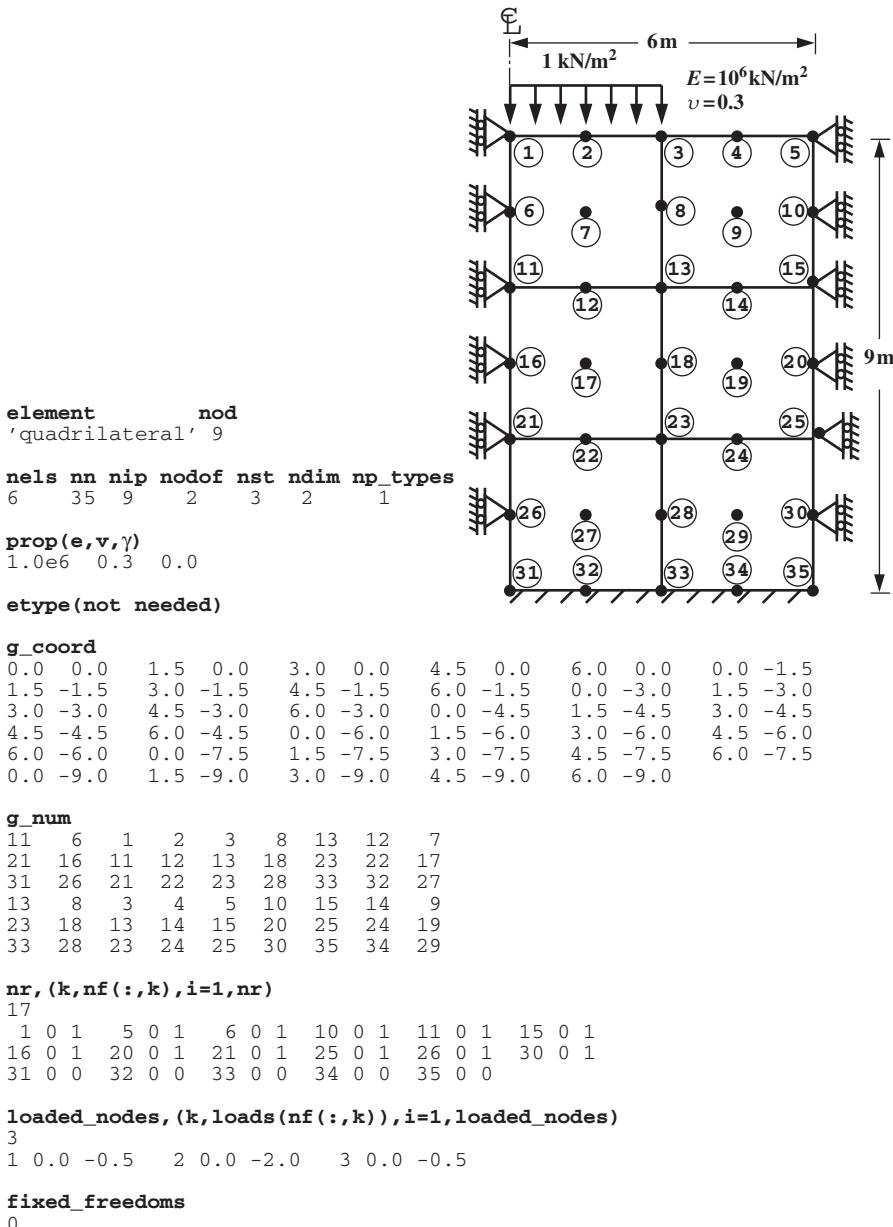


Figure 5.27 Mesh and data for first Program 5.4 example

and a more commonly available element in commercial programs is the 20-node brick. Both of these elements are available in Programs 5.3 and 5.4, as is an intermediate 14-node element proposed by Smith and Kidger (1992). This intermediate element has eight corner nodes, supplemented by six mid-face nodes, with a numbering system given in Figure 5.32. There are several versions of this element, and one of them ('Type 6') is illustrated in the next example (see Section 3.7.9).

There are 48 equations and the skyline storage is 610

```

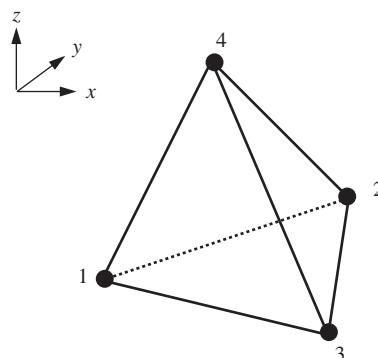
Node   x-disp      y-disp
 1    0.0000E+00 -0.5299E-05
 2   -0.4004E-06 -0.4988E-05
 3   -0.6190E-06 -0.3343E-05
 4   -0.4004E-06 -0.1697E-05
 5    0.0000E+00 -0.1387E-05
 6    0.0000E+00 -0.4307E-05
 7    0.1856E-06 -0.3911E-05
 8    0.3167E-06 -0.2786E-05
 9    0.1856E-06 -0.1661E-05
10    0.0000E+00 -0.1264E-05
.
.
.
31   0.0000E+00  0.0000E+00
32   0.0000E+00  0.0000E+00
33   0.0000E+00  0.0000E+00
34   0.0000E+00  0.0000E+00
35   0.0000E+00  0.0000E+00

```

The integration point (nip= 9) stresses are:

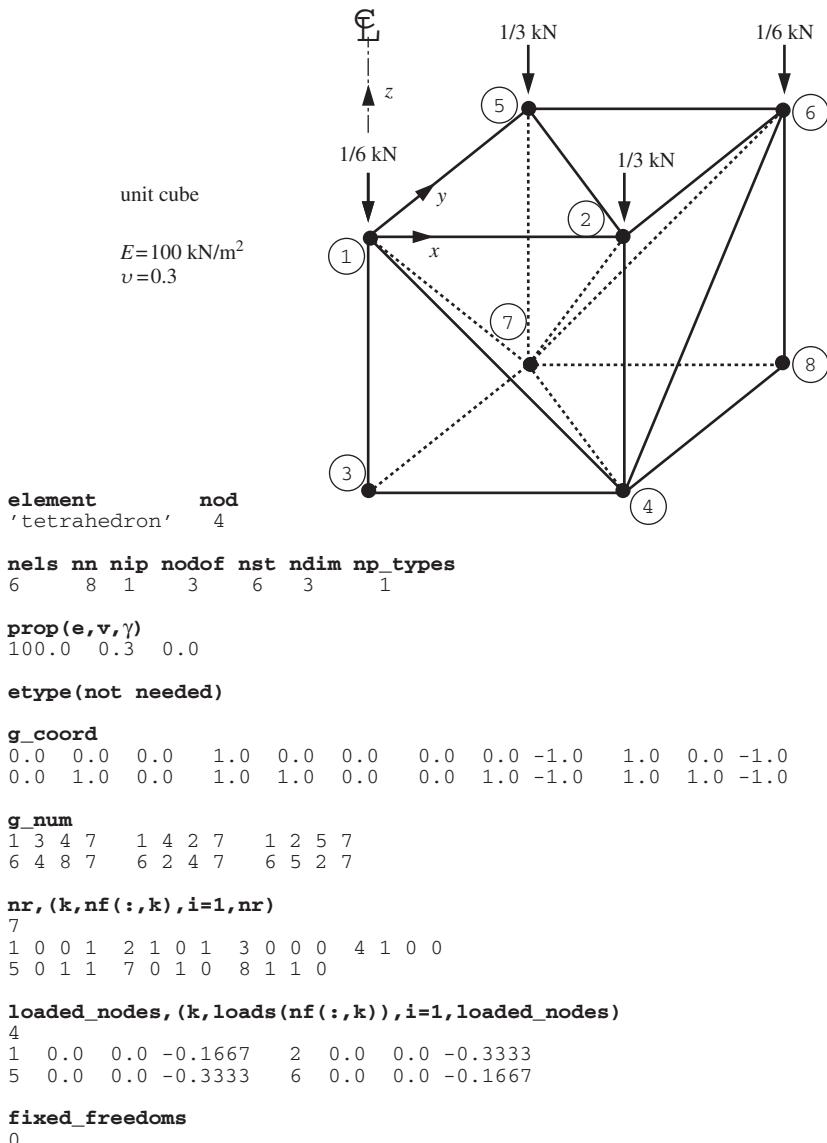
Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.3381E+00	-0.3381E+00	-0.6323E+00	-0.1035E+01	-0.4108E-01
1	0.1500E+01	-0.3381E+00	-0.5703E+00	-0.1047E+01	0.5463E-01
1	0.2662E+01	-0.3381E+00	-0.3507E+00	-0.6910E+00	0.1917E+00
1	0.3381E+00	-0.1500E+01	-0.2434E+00	-0.9108E+00	0.2695E-01
1	0.1500E+01	-0.1500E+01	-0.2597E+00	-0.8766E+00	0.1085E+00
1	0.2662E+01	-0.1500E+01	-0.1681E+00	-0.5906E+00	0.2173E+00
1	0.3381E+00	-0.2662E+01	-0.1395E+00	-0.9083E+00	0.5062E-01
1	0.1500E+01	-0.2662E+01	-0.1846E+00	-0.8070E+00	0.1512E+00
1	0.2662E+01	-0.2662E+01	-0.1714E+00	-0.5698E+00	0.2648E+00
6	0.4500E+01	-0.6338E+01	-0.2436E+00	-0.4162E+00	0.4191E-01
6	0.5662E+01	-0.6338E+01	-0.2740E+00	-0.3928E+00	0.1298E-01
6	0.3338E+01	-0.7500E+01	-0.2245E+00	-0.4855E+00	0.4446E-01
6	0.4500E+01	-0.7500E+01	-0.2283E+00	-0.4382E+00	0.2936E-01
6	0.5662E+01	-0.7500E+01	-0.2449E+00	-0.4208E+00	0.8179E-02
6	0.3338E+01	-0.8662E+01	-0.2137E+00	-0.4886E+00	0.3231E-01
6	0.4500E+01	-0.8662E+01	-0.2059E+00	-0.4572E+00	0.2255E-01
6	0.5662E+01	-0.8662E+01	-0.2063E+00	-0.4448E+00	0.6023E-02

**Figure 5.28** Results from first Program 5.4 example



Freedoms numbered from 1-12 in same order as the nodes

**Figure 5.29** Local node and freedom numbering for the 4-node tetrahedral element



**Figure 5.30** Mesh and data for second Program 5.4 example

The analysis shown in Figure 5.33 is of a ‘patch’ mesh suggested by Peano (1987) for testing the admissibility of solid elements. The outer cube has smooth, rigid boundary conditions to the left ( $x = 0$ ) and bottom ( $z = -1$ ), and the five nodes on the front face of the exterior box are given a uniform  $y$ -displacement of 0.01. The results are shown in Figure 5.34, where it can be seen that a completely homogeneous strain field has resulted in a stress of  $\sigma_y = -0.01 \text{ kN/m}^2$  at all integrating points within the mesh. Thus, the element passes the patch test. The current example used  $nip=27$ , but users could experiment with different orders of integration.

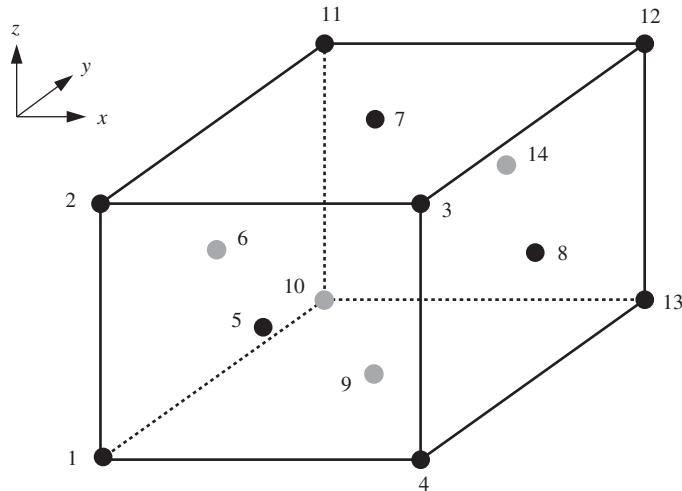
There are 12 equations and the skyline storage is 69

Node	x-disp	y-disp	z-disp
1	0.0000E+00	0.0000E+00	-0.1000E-01
2	0.3000E-02	0.0000E+00	-0.1000E-01
3	0.0000E+00	0.0000E+00	0.0000E+00
4	0.3000E-02	0.0000E+00	0.0000E+00
5	0.0000E+00	0.3000E-02	-0.9999E-02
6	0.3000E-02	0.3000E-02	-0.1000E-01
7	0.0000E+00	0.3000E-02	0.0000E+00
8	0.3000E-02	0.3000E-02	0.0000E+00

The integration point (nip= 1) stresses are:

Element	x-coord	y-coord	z-coord	sig_x	sig_y	sig_z	tau_xy	tau_yz	tau_zx
1	0.2500E+00	0.2500E+00	-0.7500E+00	-0.1965E-04	-0.2100E-04	-0.1000E+01	0.0000E+00	0.0000E+00	0.0000E+00
2	0.5000E+00	0.2500E+00	-0.5000E+00	0.2302E-05	0.1389E-04	-0.1000E+01	-0.6475E-05	0.2974E-04	0.2326E-04
3	0.2500E+00	0.5000E+00	-0.2500E+00	0.1230E-04	0.2075E-05	-0.9999E+00	0.0000E+00	0.3641E-04	0.2974E-04
4	0.7500E+00	0.7500E+00	-0.7500E+00	-0.1087E-04	-0.9023E-05	-0.1000E+01	0.1556E-05	-0.7467E-05	-0.9316E-05
5	0.7500E+00	0.5000E+00	-0.5000E+00	0.6102E-05	-0.1297E-05	-0.1000E+01	-0.8752E-05	-0.2541E-04	-0.3189E-04
6	0.5000E+00	0.7500E+00	-0.2500E+00	0.9809E-05	0.1536E-04	-0.9999E+00	0.2158E-05	-0.3632E-04	-0.4299E-04
7									
8									
9									
10									
11									
12									
13									
14									

**Figure 5.31** Results from second Program 5.4 example



Freedoms numbered from 1-42 in same order as the nodes

**Figure 5.32** Local node and freedom numbering for the 14-node hexahedral element

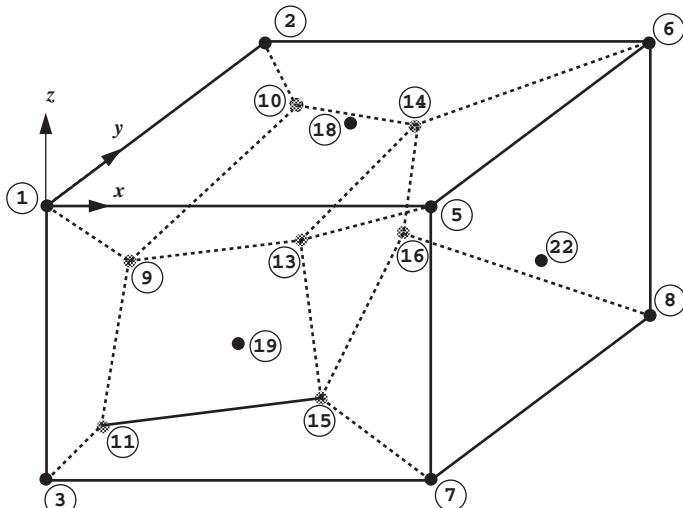


Figure shows visible nodes on the surface, plus corner nodes only of inner element.  
Outer 'box' is a unit cube

```
element      nod
'hexahedron' 14
```

```
nels nn nip nodof nst ndim np_types
7    40   27     3    6   3       1
```

```
prop(e,v,g)
1.0  0.49  0.0
```

```
etype(not needed)
```

```
g_coord
```

```
0.0000  0.0000  0.0000      0.0000  1.0000  0.0000
0.0000  0.0000 -1.0000      0.0000  1.0000 -1.0000
```

```
. (g_coord data for nodes 5-36 omitted here)
```

```
0.1423  0.5412 -0.8680      0.5187  0.8800 -0.8425
0.8758  0.5072 -0.8518      0.4992  0.1685 -0.8773
```

```
g_num
```

```
11    9   13   15   25   23   24   28   27   12   10   14   16   26
 3    1    9   11   33   17   29   23   37    4    2    10   12   34
 9    1    5   13   32   29   18   31   24   10    2    6   14   30
15   13   5    7   36   28   31   22   39   16   14    6    8   35
 3   11   15   7   40   37   27   39   21    4   12   16    8   38
12   10   14   16   26   34   30   35   38    4    2    6    8   20
 3    1    5    7   19   33   32   36   40   11    9   13   15   25
```

```
nr,(k,nf(:,k),i=1,nr)
```

```
10
```

```
1 0 1 1    2 0 0 1    3 0 1 0    4 0 0 0    6 1 0 1
7 1 1 0    8 1 0 0    17 0 1 1   20 1 0 1   21 1 1 0
```

```
loaded_nodes
```

```
0
```

```
fixed_freedoms,(node(i),sense(i),value(i),i=1,fixed_freedoms)
```

```
5
```

```
1 2  0.01  3 2  0.01  5 2  0.01  7 2  0.01  19 2  0.01
```

Figure 5.33 Mesh and data for third Program 5.4 example

There are 105 equations and the skyline storage is 5487

Node	x-disp	y-disp	z-disp
1	0.0000E+00	0.1000E-01	0.4900E-02
2	0.0000E+00	0.0000E+00	0.4900E-02
3	0.0000E+00	0.1000E-01	0.0000E+00
4	0.0000E+00	0.0000E+00	0.0000E+00
5	0.4900E-02	0.1000E-01	0.4900E-02
6	0.4900E-02	0.0000E+00	0.4900E-02
7	0.4900E-02	0.1000E-01	0.0000E+00
8	0.4900E-02	0.0000E+00	0.0000E+00
9	0.8085E-03	0.7020E-02	0.3651E-02
10	0.1333E-02	0.2300E-02	0.3675E-02
.	.	.	.
37	0.6973E-03	0.4588E-02	0.6468E-03
38	0.2542E-02	0.1200E-02	0.7718E-03
39	0.4291E-02	0.4928E-02	0.7262E-03
40	0.2446E-02	0.8315E-02	0.6012E-03

The integration point (nip=27) stresses are:

Element	x-coord	y-coord	z-coord	sig_x	sig_y	sig_z	tau_xy	tau_yz	tau_zx
1	0.2589E+00	0.3618E+00	-0.3200E+00	-0.1793E-07	-0.1000E-01	-0.1827E-07	0.1618E-09	-0.1391E-08	-0.2100E-09
1	0.4879E+00	0.3762E+00	-0.3354E+00	-0.5606E-08	-0.1000E-01	-0.6012E-08	-0.4430E-09	-0.1763E-08	-0.1400E-09
1	0.7170E+00	0.3905E+00	-0.3508E+00	0.1582E-08	-0.1000E-01	-0.7704E-10	-0.4557E-09	-0.8293E-09	-0.1176E-09
.	.	.	.	.	.	.	.	.	.
7	0.4889E+00	0.2946E+00	-0.5157E+00	-0.8196E-07	-0.1000E-01	-0.8207E-07	-0.2538E-09	-0.8485E-09	-0.2121E-09
7	0.7010E+00	0.2977E+00	-0.5042E+00	-0.3727E-08	-0.1000E-01	-0.3632E-08	-0.7091E-09	0.5663E-09	-0.1256E-09
7	0.3198E+00	0.3080E+00	-0.7564E+00	0.3311E-07	-0.1000E-01	0.3315E-07	0.1318E-08	0.7962E-09	-0.1938E-09
7	0.4965E+00	0.2980E+00	-0.7221E+00	0.1311E-07	-0.1000E-01	0.1332E-07	0.1924E-09	0.1619E-08	0.1072E-09
7	0.6731E+00	0.2881E+00	-0.6878E+00	-0.1021E-07	-0.1000E-01	-0.1057E-07	0.8205E-10	0.1063E-08	0.5315E-10

Figure 5.34 Results from third Program 5.4 example

## 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

```

PROGRAM p55
!-----
! Program 5.5 Plane or axisymmetric thermal strain analysis of an elastic
! rectangular solid using 3-, 6-, 10- or 15-node right-angled
! triangles or 4-, 8- or 9-node rectangular quadrilaterals.
! Mesh numbered in x(r)- or y(z)- direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,ndim=2,ndof,nels,neq,nip, &
nlen,nn,nod,nodof=2,nprops=4,np_types,nr,nspr,nst=3,nxe,nye
REAL(iwp)::det,gtemp,one=1.0_iwp,penalty=1.0e20_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,dir,element,type_2d
!-----dynamic arrays-----

```

```

INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:,g_num(:,:)),kdiag(:,nf(:,:, &
no(:,node(:,num(:,sense(:,sno(:,spno(:,spse(:)

REAL(iwp),ALLOCATABLE::bee(:,:,coord(:,:,dee(:,:,der(:,:,deriv(:,:), &
dtel(:),dtemp(:),eld(:),eps(:),etl(:),fun(:),gc(:,g_coord(:,:), &
jac(:,:,km(:,:,kv(:),loads(:),points(:,:,prop(:,:,sigma(:),spva(:),&
teps(:),tload(:),value(:),weights(:),x_coords(:),y_coords(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,element,nod,dir,nxe,nye,nip,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); ndof=nod*nodof
IF(type_2d=='axisymmetric') nst=4
ALLOCATE(nf(ndof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),fun(nod),&
coord(nod,ndim),jac(ndim,ndim),g_num(nod,nels),der(ndim,nod), &
deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),eld(ndof),weights(nip), &
g_g(ndof,nels),prop(nprops,np_types),num(nod),x_coords(nxe+1), &
y_coords(nye+1),etype(nels),gc(ndim),dee(nst,nst),sigma(nst),eps(nst), &
teps(nst),etl(ndof),dtel(nod),dtemp(nn))

READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
READ(10,*)dtemp; ALLOCATE(loads(0:neq),tload(0:neq),kdiag(neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1
CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))') &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----element stiffness integration and assembly-----
CALL sample(element,points,weights)
kv=zero; gc=one; teps=zero; tload=zero
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel);
    dtel=dtemp(num); etl=zero; km=zero
    int_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        IF(type_2d=='axisymmetric') THEN
            gc=MATMUL(fun,coord); bee(4,1:ndof-1:2)=fun(:)/gc(1)
        END IF
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)*gc(1)
        gtemp=dot_product(fun,dtel); teps(1:2)=gtemp*prop(3:4,etype(iel))
        etl=etl+MATMUL(MATMUL(TRANSPOSE(bee),dee),teps)*det*weights(i)*gc(1)
    END DO int_pts_1
    tload(g)=tload(g)+etl; CALL fsparv(kv,km,g,kdiag)
END DO elements_2
READ(10,*)nspr
IF(nspr/=0)THEN; ALLOCATE(sno(nspr),spno(nspr),spse(nspr),spva(nspr))
    READ(10,*)(spno(i),spse(i),spva(i),i=1,nspr)
    DO i=1,nspr; sno(i)=nf(spse(i),spno(i)); END DO
    kv(kdiag(sno))=kv(kdiag(sno))+spva
END IF

```

```

loads=zeros; READ(10,*) loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
loads=loads+tload; READ(10,*) fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),
              value(fixed_freedoms),no(fixed_freedoms)) &
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; loads(no)=kv(kdiag(no))*value
END IF
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag); loads(0)=zero
IF(type_2d=='axisymmetric')THEN
    WRITE(11,'(/A)')" Node r-disp z-disp"; ELSE
    WRITE(11,'(/A)')" Node x-disp y-disp"
END IF
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(nf(:,k)); END DO
!-----recover stresses at nip integrating points-----
nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
CALL sample(element,points,weights)
WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
IF(type_2d=='axisymmetric')THEN
    WRITE(11,'(A,A)')" Element r-coord z-coord",
    " sig_r sig_z tau_rz sig_t"; ELSE
    WRITE(11,'(A,A)')" Element x-coord y-coord",
    " sig_x sig_y tau_xy" &
END IF
elements_3: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g);
    dtel=dtemp(num)
    int_pts_2: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        IF(type_2d=='axisymmetric')THEN
            gc=MATMUL(fun,coord); bee(4,1:ndof-1:2)=fun(:)/gc(1)
        END IF
        gtemp=dot_product(fun,dtel); teps(1:2)=gtemp*prop(3:4,etype(iel))
        eps=MATMUL(bee,eld)-teps; sigma=MATMUL(dee,eps)
        WRITE(11,'(I5,6E12.4)')iel,gc,sigma
    END DO int_pts_2
END DO elements_3
CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p55

```

Program 5.5 is a modified version of Program 5.1 that computes elastic deformations and stresses due to temperature changes in addition to more conventional loading. New data involves the thermal expansion coefficients  $\alpha_x$  and  $\alpha_y$ , which may be anisotropic, and the temperature *change* to be applied at all nodes in the mesh. The expansion coefficients are read into the *prop* array and the temperature changes into the *dtemp* vector. In addition, this program includes the option of adding a discrete spring to any freedom in the mesh, where scalar *nspr* represents the number of springs. The *spno*, *spse* and *spva* vectors hold, respectively, the node, the sense and the value of the spring stiffness. If no springs are required, *nspr* is simply read as zero and no further spring data is needed.

As described in Section 2.10.1 and (2.72), the nodal thermal changes in each element  $\text{dte1}$  are integrated using the Gauss points to generate element thermal forces  $\text{et1}$  which are then assembled into a global thermal force vector  $\text{tload}$ . The global thermal force vector is then added to any additional loading and the global equilibrium equations solved in the usual way for the global nodal displacements.

The Gauss point stresses are retrieved at the element level by computing the total strains within each element, using  $\text{MATMUL}(\text{bee}, \text{eld})$ , and then *subtracting* the thermal strains  $\text{teps}$ . Any remaining strains  $\text{eps}$  after subtraction are multiplied by the constitutive matrix  $\text{dee}$  to give the stresses  $\text{sigma}$ .

The example and data shown in Figure 5.35 represent an initially unstressed slender cantilever beam of length  $L = 1$  and depth  $2b$ , where  $b = 0.05$  ( $I = 8.333 \times 10^{-5}$ ). The mesh consists of 40 square 8-node elements with reduced integration, arranged in two rows and 20 columns, with node and freedom numbering in the ‘y’ direction. A few nodes are shown numbered for clarification.

The beam is supported at its tip by a spring of stiffness  $K = 50$ , and subjected to a linear temperature change variation  $T$  through its depth given by

$$T = \frac{\Delta Ty}{b} \quad (5.8)$$

where  $\Delta T = 0.5$ .

The thermal expansion coefficient is isotropic and given by  $\alpha = \alpha_x = \alpha_y = 1 \times 10^{-5}$ . Young’s modulus is set to  $E = 1 \times 10^5$ , and Poisson’s ratio to  $\nu = 0$ .

The data requires the temperature change to be read in at all 165 nodes, which varies linearly across the depth from 0.5 on the top flange, to  $-0.5$  at the bottom. The only spring in this example has a stiffness of 50.0 and is attached to the second freedom (y-direction) of node 163 on the neutral axis of the beam tip. There are no additional loads or fixed displacements applied in this example.

The analytical solution based on slender beam theory (e.g., Johns, 1965) can be rearranged to predict a tip displacement  $\delta_t$  given by

$$\delta_t = \frac{3\alpha\Delta TL^2EI}{4b(3EI + KL^3)} \quad (5.9)$$

For the particular case considered here, (5.9) gives  $\delta_t = 1.67 \times 10^{-5}$  which is in good agreement with the computed vertical deflection of  $1.66 \times 10^{-5}$  at node 163, as given in the results file of Figure 5.36.

The stresses generated in the beam follow from standard beam theory as

$$\sigma_x = \frac{3P(L-x)y}{2b^3} \quad (5.10)$$

where  $P = K\delta_t$  is the spring force, and  $x$  and  $y$  are, respectively, the distance from the cantilever support and the neutral axis. The program computes stresses at the element centroids, so if we consider either of the elements closest to the cantilever support where  $x = y = 0.025$ , equation (5.10) gives  $\sigma_x = 0.24$ , which is again in close agreement with the computed values in elements 1 and 2 as shown in Figure 5.36.

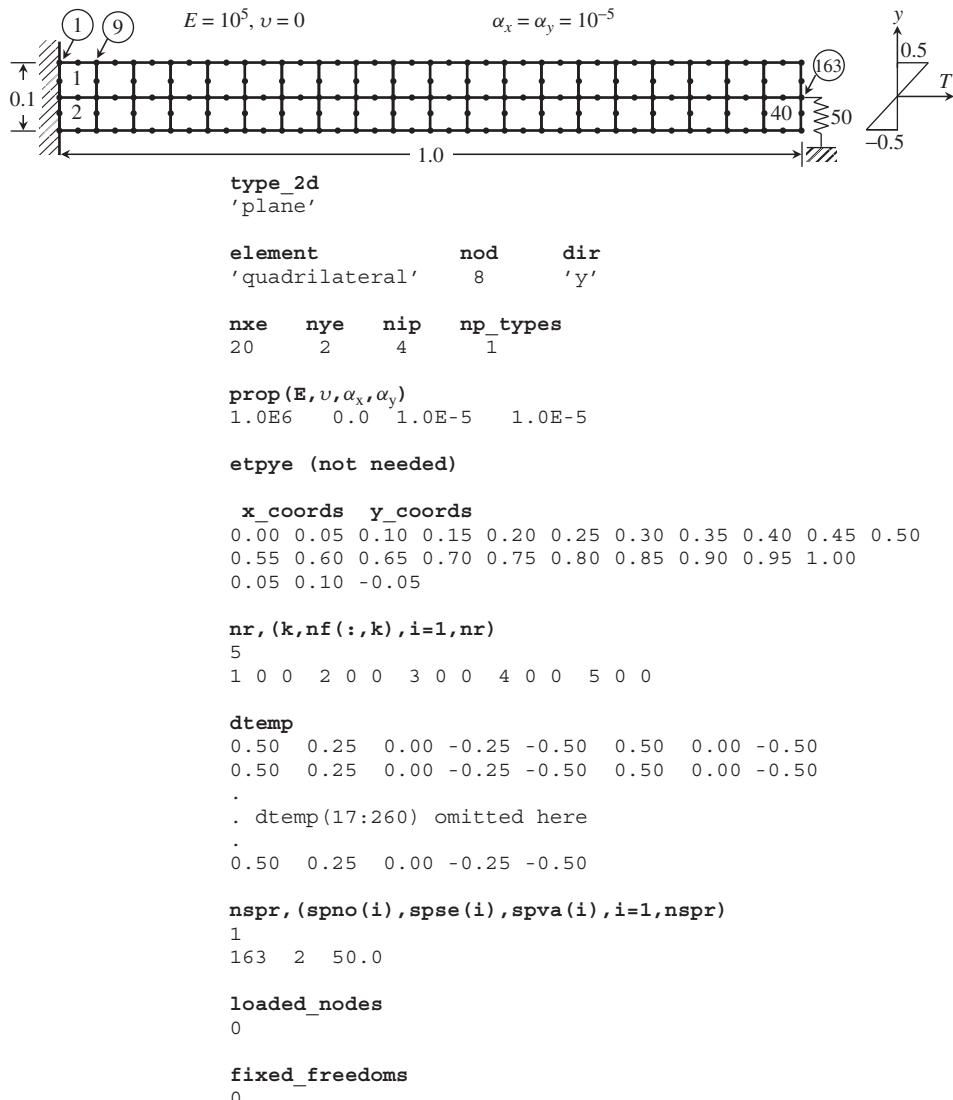


Figure 5.35 Mesh and data for Program 5.5 example

**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**

PROGRAM p56

```

! -----
! Program 5.6 Three-dimensional strain of an elastic solid using
! 8-, 14- or 20-node brick hexahedra. Mesh numbered in x-z
! planes then in the y-direction. No global stiffness matrix

```

```

!           assembly. Diagonally preconditioned conjugate gradient solver.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15),npri=1,nstep=1
INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,k,loaded_nodes,ndim=3,      &
ndof,nels,neq,nip,nlen,nn,nprops=2,np_types,nod,nodof=3,nr,nst=6,nxe,      &
nye,nze
REAL(iwp)::alpha,beta,cg_tol,det,dtim,one=1.0_iwp,penalty=1.0e20_iwp,up,  &
zero=0.0_iwp; CHARACTER(LEN=15)::argv,element='hexahedron'
LOGICAL::cg_converged,solid=.TRUE.

!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:)),g_num(:,:),nf(:,:,no(:),      &
node(:),num(:),sense(:)
REAL(iwp),ALLOCATABLE::bee(:,:,coord(:,:,d(:,:,dee(:,:,der(:,:),
& deriv(:,:,diag_precon(:,:),eld(:,:),fun(:,:),gc(:,:),g_coord(:,:),jac(:,:,,
& km(:,:,loads(:,:),p(:,:),points(:,:),prop(:,:,sigma(:,:),store(:,:,
& storkm(:,:,u(:,:),weights(:,:),x(:,:),xnew(:,:),x_coords(:,:,
y_coords(:,:),z_coords(:,:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nod,nxe,nye,nze,nip,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye,nze); ndof=nod*nodof
ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),coord(nod,ndim),      &
jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),fun(nod),gc(ndim),      &
bee(nst,ndof),km(ndof,ndof),eld(ndof),sigma(nst),g_coord(ndim,nn),      &
g_num(nod,nels),weights(nip),num(nod),g_g(ndof,nels),x_coords(nxe+1),      &
g(ndof),y_coords(nye+1),z_coords(nze+1),etype(nels),      &
prop(nprops,np_types),storkm(ndof,nodof,nels))

READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,z_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formmf(nf); neq=MAXVAL(nf)
WRITE(11,'(A,I10,A)')" There are",neq," equations"
ALLOCATE(p(0:neq),loads(0:neq),x(0:neq),xnew(0:neq),u(0:neq),      &
diag_precon(0:neq),d(0:neq))
CALL sample(element,points,weights); diag_precon=zero

!-----element stiffness integration, storage and preconditioner-----
elements_1: DO iel=1,nels
    CALL hexahedron_xz(iel,x_coords,y_coords,z_coords,coord,num)
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); g=g_g(:,iel); coord=TRANSPOSE(g_coord(:,num)); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1
    storkm(:,:,iel)=km
    DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
END DO elements_1

!-----invert the preconditioner and get starting loads--
loads=zero; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),      &
value(fixed_freedoms),no(fixed_freedoms),store(fixed_freedoms))

```

```

READ(10,*) (node(i),sense(i),value(i),i=1,fixed_freedoms)
DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
diag_precon(no)=diag_precon(no)+penalty; loads(no)=diag_precon(no)*value
store=diag_precon(no)
END IF
CALL mesh_ensi(argv,nlen,g_coord,g_num,element,etype,nf,loads(1:),      &
              nstep,npri,dtim,solid)
diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
pcg: DO; cg_iters=cg_iters+1; u=zero
elements_2: DO iel=1,nels
  g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
END DO elements_2
IF(fixed_freedoms/=0)u(no)=p(no)*store; up=DOT_PRODUCT(loads,d)
alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha; loads=loads-u*alpha
d=diag_precon*loads; beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
CALL checon(xnew,x,cg_tol,cg_converged)
IF(cg_converged.OR.cg_iters==cg_limit)EXIT
END DO pcg
WRITE(11,'(A,I5)')" Number of cg iterations to convergence was",cg_iters
WRITE(11,'(/A)')" Node x-disp y-disp z-disp"; loads=xnew
DO k=1,nn; WRITE(11,(I6,3E12.4)')k,loads(nf(:,k)); END DO
CALL dismsh_ensi(argv,nlen,nstep,nf,xnew(1:))
!-----recover stresses at nip integrating point-----
nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
CALL sample(element,points,weights); loads(0)=zero
WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
WRITE(11,'(A,,A)')" Element x-coord y-coord z-coord",   &
" sig_x sig_y sig_z tau_xy tau_yz tau_zx"
elements_3: DO iel=1,nels
  CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel))); num=g_num(:,iel)
  coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
  gauss_pts_2: DO i=1,nip
    CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
    gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    sigma=MATMUL(dee,MATMUL(bee,eld)); WRITE(11,'(I8,4X,3E12.4)')iel,gc
    WRITE(11,'(6E12.4)')sigma
  END DO gauss_pts_2
END DO elements_3
STOP
END PROGRAM p56

```

Section 3.5 described an ‘element-by-element’ approach to the solution of linear static equilibrium problems in which the equation solution process could be carried out by the pcg technique without ever assembling element matrices into a global (stiffness) matrix. The essential process was described by equations (3.20) to (3.23). Program 5.6 will now be used to solve once more the problem illustrated in Figure 5.24 and previously solved using an assembly technique by Program 5.3. A structure chart for the pcg algorithm is shown in Figure 5.37.

All of the elements are looped in order to compute their stiffness matrices, which are stored in the array `storkm` for use later in the pcg solution algorithm. This loop (called `elements_2`) also builds the preconditioning matrix, which is simply the inverse of the diagonal terms in what would have been the assembled global stiffness matrix.

There are 320 equations and the skyline storage is 5292

Node	x-disp	y-disp
1	0.0000E+00	0.0000E+00
2	0.0000E+00	0.0000E+00
3	0.0000E+00	0.0000E+00
4	0.0000E+00	0.0000E+00
5	0.0000E+00	0.0000E+00
6	-0.9089E-09	0.7926E-07
7	-0.9277E-21	-0.4382E-07
8	0.9089E-09	0.7926E-07
.	.	.
161	0.2511E-05	-0.1650E-04
162	0.1253E-05	-0.1659E-04
163	-0.2288E-19	-0.1661E-04
164	-0.1253E-05	-0.1659E-04
165	-0.2511E-05	-0.1650E-04

The integration point (nip= 1) stresses are:

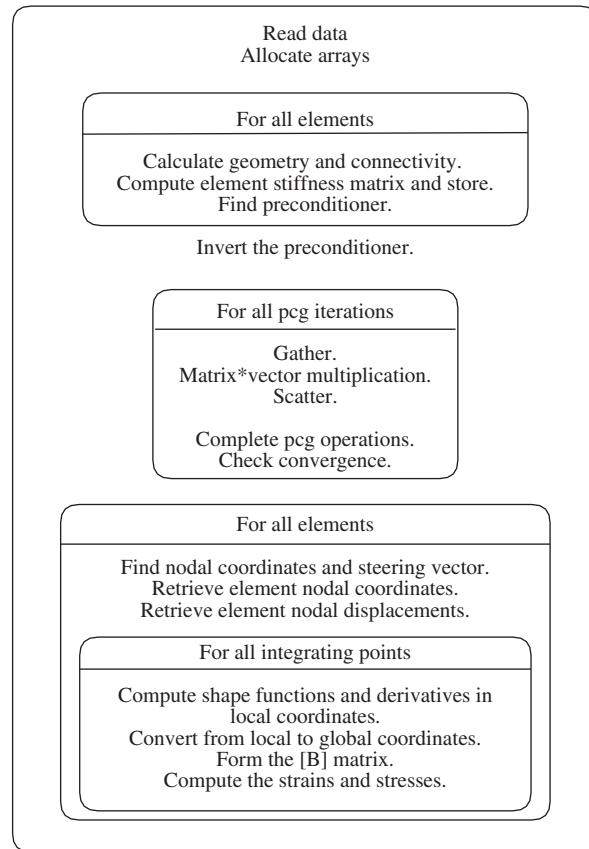
Element	x-coord	y-coord	sig_x	sig_y	tau_xy
1	0.2500E-01	0.2500E-01	-0.2372E+00	-0.3837E-02	-0.5421E-02
2	0.2500E-01	-0.2500E-01	0.2372E+00	0.3837E-02	-0.5421E-02
3	0.7500E-01	0.2500E-01	-0.2353E+00	0.3186E-02	0.1133E-01
4	0.7500E-01	-0.2500E-01	0.2353E+00	-0.3186E-02	0.1133E-01
.	.	.	.	.	.
39	0.9750E+00	0.2500E-01	-0.1084E-01	0.3072E-02	0.7027E-02
40	0.9750E+00	-0.2500E-01	0.1084E-01	-0.3072E-02	0.7027E-02

**Figure 5.36** Results from Program 5.5 example

The preconditioning matrix (stored as a vector) is called `diag_precon`. The section commented ‘pcg equation solution’ carries out the vector operations described in equations (3.22) within the iterative loop labelled `pcg`. The matrix–vector multiply needed in the first of (3.22) is done using (3.23). The steering vector `g` ‘gathers’ the appropriate components of `p`, to be multiplied by the element stiffness matrix `km` retrieved from `storkm`. Similarly, the vector `g` ‘scatters’ the result of the matrix–vector multiply to appropriate locations in `u`. A tolerance `pcg_tol` enables the iterations to be stopped when successive solutions are ‘close enough’, but since `pcg` is a loop which might carry on ‘forever’, an iteration ceiling, `cg_limit`, is specified also. Strains and stresses can then be recovered from the displacements in the usual manner.

The data for the program are shown in Figure 5.38. The only changes from Figure 5.24, which used an assembly strategy, are the two additional data values read in for `cg_tol` and `cg_limit`, which are set to be  $1 \times 10^{-5}$  and 200, respectively.

Figure 5.39 shows the results which may be compared with those listed in Figure 5.25. The specified accuracy of solution took 64 iterations in this case. Since, in perfect arithmetic, the conjugate gradient process should converge in at most `neq` iterations (124 in this case), the amount of computational effort in large problems may seem to be daunting. Fortunately, as the set of equations to be solved grows larger, the proportion of iterations to converge to the number of equations (`cg_iters/neq`) usually decreases dramatically. It does, however, depend on the ‘condition number’ (nature of the eigenvalue spectrum) of



**Figure 5.37** Structure chart for pcg algorithm as used in Program 5.6. No global matrix assembly

the assembled stiffness matrix. With no diagonal preconditioning (`diag_precon=1.0`), the number of iterations for convergence rises to 70.

The program also introduces the subroutines `mesh_ensi` and `dismsh_ensi`, which generate output files that can be used with ParaView (see Section 1.11).

### 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

PROGRAM p57

```

!-----
! Program 5.7 Three-dimensional strain of an elastic solid using
!           8-, 14- or 20-node brick hexahedra. Mesh numbered in x-z
!           planes then in the y-direction. No global stiffness matrix.
!           Diagonally preconditioned conjugate gradient solver.
!           Optimised maths library, Abaqus UMAT version.
!-----

```

```

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15);npri=1,nstep=1
INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,k,loaded_nodes,ndim=3,  &
ndof,nels,ned,nip,nn,nprops=2,np_types,nod,nodof=3,nr,nst=6,nxe,nye,  &
nze,nlen,step,idamax
REAL(iwp)::alpha,beta,big,cg_tol,det,one=1.0_iwp,penalty=1.0e20_iwp,up,  &
zero=0.0_iwp,dtim=0.0_iwp,ddot,maxload,maxdiff
CHARACTER(LEN=15)::element='hexahedron',argv
LOGICAL::cg_converged,solid=.TRUE.

!-----variables required by UMAT-----
!ntens == nst; noel == iel; nprops == npops
INTEGER,PARAMETER::ndi=3; INTEGER::nshr,nstatv,npt,layer,kspt,kstep,kinc
REAL(iwp)::sse,spd,scd,rpl,dtim,temp,dtemp,pnewdt,celent
CHARACTER(LEN=80)::cmname
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:),g_g(:, :, ),g_num(:, :, ),nf(:, :, ),no(:, ),  &
node(:, ),num(:, ),sense(:, ))
REAL(iwp),ALLOCATABLE::bee(:, :, ),coord(:, :, ),d(:, ),dee(:, :, ),der(:, :, ),gc(:, ),&
deriv(:, :, ),diag_precon(:, ),eld(:, ),fun(:, ),g_coord(:, :, ),g_pmul(:, :, ),  &
g_utemp(:, :, ),jac(:, :, ),km(:, :, ),loads(:, ),p(:, ),points(:, :, ),prop(:, :, ),  &
sigma(:, ),store(:, ),storkm(:, :, :, ),u(:, ),value(:, ),weights(:, ),x(:, ),xnew(:, ),&
x_coords(:, ),y_coords(:, ),z_coords(:, ),oldlds(:, ),timest(:, )
!-----dynamic arrays required by UMAT-----
! stress(ntens) == sigma(nst); ddsdde(ntens,ntens) == dee(nst,nst)
! dstran(ntens) == eld(nst); coords() == points()
REAL(iwp),ALLOCATABLE::statev(:, ),ddsdde(:, ),drplde(:, ),drpldt(:, ),stran(:, ),&
time(:, ),predef(:, ),dpred(:, ),props(:, ),drot(:, :, ),dfgrd0(:, :, ),dfgrd1(:, :, )
!-----input and initialisation-----
ALLOCATE(timest(2)); timest=zero; timest(1)=elap_time()
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nod,nxe,nye,nze,nip,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye,nze); nodof=nod*nodof
ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),coord(nod,ndim),  &
jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),fun(nod),gc(ndim),  &
bee(nst,nodof),km(ndof,ndof),eld(ndof),sigma(nst),g_coord(ndim,nn),  &
g_num(nod,nels),weights(nip),num(nod),g_g(ndof,nels),x_coords(nxe+1),  &
g(ndof),y_coords(nye+1),z_coords(nze+1),etype(nels),g_pmul(ndof,nels),  &
prop(npops,np_types),storkm(ndof,ndof,nels),g_utemp(ndof,nels))
ALLOCATE(props(npops)) ! Abaqus UMAT
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,z_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(p(0:neq),loads(0:neq),x(0:neq),xnew(0:neq),u(0:neq),  &
diag_precon(0:neq),d(0:neq),oldlds(0:neq))
diag_precon=zero; CALL sample(element,points,weights)
timest(2)=elap_time()
!-----element stiffness integration, storage and preconditioner-----
elements_1: DO iel=1,nels
    CALL hexahedron_xz(iel,x_coords,y_coords,z_coords,coord,num)
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
    props(1)=prop(1,etype(iel)); props(2)=prop(2,etype(iel)); dee=zero
    CALL umat(sigma,statev,dee,sse,spd,scd,rpl,ddsdde,drplde,  &
drpldt,stran,eld,time,dtim,temp,dtemp,predef,dpred,cmname,ndi,nshr,  &
nst,nstatv,props,npops,points,drot,pnewdt,celent,dfgrd0,dfgrd1,iel,  &
npt,layer,kspt,kstep,kinc)
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    gauss_pts_1: DO i=1,nip

```

```

CALL shape_der(der,points,i); jac=MATMUL(der,coord)
det=determinant(jac); CALL invert(jac)
deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
END DO gauss_pts_1; storkm(:,:,iel)=km
DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
END DO elements_1
!-----invert the preconditioner and get starting loads--
loads=zeros; READ(10,*).loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
oldlds = loads ; READ(10,*).fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),
             &
             value(fixed_freedoms),no(fixed_freedoms),store(fixed_freedoms))
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    diag_precon(no)=diag_precon(no)+penalty; loads(no)=diag_precon(no)*value
    store=diag_precon(no)
END IF
CALL mesh_ensi(argv,nlen,g_coord,g_num,element,etype,nf,oldlds(1:),      &
              nstep,npri,dtim,solid)
diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
pcg: DO
    cg_iters=cg_iters+1; u=zero; g_utemp=zero
    elements_2: DO iel=1,nels; g_pmul(:,iel)=p(g_g(:,iel))
    END DO elements_2
    CALL dsymm('L','1',ndof,nels,one,km,ndof,g_pmul,ndof,one,g_utemp,ndof)
    elements_2a: DO iel=1,nels; u(g_g(:,iel))=u(g_g(:,iel))+g_utemp(:,iel)
    END DO elements_2a
    IF(fixed_freedoms/=0) u(no)=p(no)*store; up=DDOT(neq,loads,1,d,1)
    alpha=up/DDOT(neq,p,1,u,1); CALL daxpy(neq,alpha,p,1,xnew,1)
    alpha=-alpha; CALL daxpy(neq,alpha,u,1,loads,1); d=diag_precon*loads
    beta=DDOT(neq,loads,1,d,1)/up; p=d+p*beta
    CALL checon(xnew,x,cg_tol,cg_converged)
    IF(cg_converged.OR.cg_iters==cg_limit)EXIT
END DO pcg
WRITE(11,'(/A)')" Node x-disp y-disp z-disp"
DO k=1,nn; WRITE(11,'(I10,3E12.4)')k,xnew(nf(:,k)); END DO
CALL dismsh_ensi(argv,nlen,nstep,nf,xnew(1:))
!-----recover stresses at nip integrating point-----
nip=1; DEALLOCATE(points,weights); ALLOCATE(points(nip,ndim),weights(nip))
WRITE(11,'(/A,I2,A)')" The integration point (nip=",nip,") stresses are:"
WRITE(11,'(A,/A)')" Element x-coord y-coord z-coord",      &
" sig_x sig_y sig_z tau_xy tau_yz tau_zx"
CALL sample(element,points,weights); xnew(0)=zero
elements_4: DO iel=1,nels
    props(1)=prop(1,etype(iel)); props(2)=prop(2,etype(iel)); dee=zero
    CALL umat(sigma,statev,dee,sse,spd,scd,rpl,ddssdt,drplde,      &
              drpldt,stran,eld,time,dtime,temp,dtemp,predef,dpred,cname,ndi,nshr, &
              nst,nstat,props,nprops,points,drot,pnewdt,celent,dfgrd0,dfgrd1,iel, &
              npt,layer,kspt,kstep,kinc)
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    g=g_g(:,iel); eld=xnew(g)
    gauss_pts_2: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        sigma=MATMUL(dee,MATMUL(bee,eld))
    END DO gauss_pts_2
END DO elements_4

```

```

      WRITE(11,'(I8,4X,3E12.4)')iel,gc; WRITE(11,'(6E12.4)')sigma
    END DO gauss_pts_2
  END DO elements_4
  WRITE(11,'(/A,F12.4,A)')" Time for setup was           ", &
                           timest(2)-timest(1),"s"
  WRITE(11,'(A,F12.4,A)')" Total time for this analysis was      ", &
                           elap_time()-timest(1),"s"
STOP
END PROGRAM p57

```

This program is used to solve a high-resolution version of a problem similar to that considered with Programs 5.4 and 5.6 and uses the ‘element-by-element’ strategy of Program 5.6. It demonstrates the ease of using external libraries and the benefits of using vendor specific tools to help reduce computation times. Figure 5.40 shows an extract of the data used in Program 5.7, which unlike those considered previously in this chapter, includes only one material type, so np\_types=1. Figure 5.41 shows a plot of the deformed mesh.

A simple change is the replacement of the book subroutine `deemat` with the external ABAQUS subroutine `umat.f` (see Appendix G) shown below (see Section 1.10.3 for an overview). The file `umat.f` needs to be compiled using a compiler flag for double precision, for example, `g95 -r8 -c umat.f` in `g95`. This highlights the potential to extend the range of material models using third party subroutines. Here the UMAT contains FORTRAN77 code to build the `dee` matrix. It could contain software for a more sophisticated stress-strain law. Despite the long list of arguments, only a few

```

nod
20

nxe nye nze nip cg_tol cg_limit np_types
1     3     2     8   1.0e-5   200      2

prop(e,v)
100.0  0.3
50.0   0.3

etype
1 2 1 2 1 2

x_coords, y_coords, z_coords
0.0   0.5
0.0   1.0   2.0   3.0
0.0  -1.0  -2.0

nr,(k,nf(:,k),i=1,nr)
46
1 0 0 1   2 1 0 1   3 1 0 1   4 0 0 1   5 1 0 1   6 0 0 1
7 1 0 1   8 1 0 1   9 0 0 1   10 1 0 1  11 0 0 0  12 0 0 0
13 0 0 0  14 0 1 1  16 0 1 1  18 0 0 0  19 0 0 0  20 0 1 1
23 0 1 1  25 0 1 1  28 0 1 1  30 0 0 0  31 0 0 0  32 0 0 0
33 0 1 1  35 0 1 1  37 0 0 0  38 0 0 0  39 0 1 1  42 0 1 1
44 0 1 1  47 0 1 1  49 0 0 0  50 0 0 0  51 0 0 0  52 0 1 1
54 0 1 1  56 0 0 0  57 0 0 0  58 0 1 1  61 0 1 1  63 0 1 1
66 0 1 1  68 0 0 0  69 0 0 0  70 0 0 0

loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
8
1  0.0  0.0  0.0417  2  0.0  0.0 -0.1667  3  0.0  0.0  0.0417
14 0.0  0.0 -0.1667 15 0.0  0.0 -0.1667 20 0.0  0.0  0.0417
21 0.0  0.0 -0.1667 22 0.0  0.0  0.0417

fixed_freedoms
0

```

**Figure 5.38** Data for Program 5.6 example

```

There are 124 equations
Number of cg iterations to convergence was 64

Node   x-disp      y-disp      z-disp
 1  0.0000E+00  0.0000E+00 -0.2246E-01
 2  0.1584E-02  0.0000E+00 -0.2255E-01
 3  0.3220E-02  0.0000E+00 -0.2333E-01
 4  0.0000E+00  0.0000E+00 -0.1849E-01
 5  0.1544E-02  0.0000E+00 -0.1884E-01
 6  0.0000E+00  0.0000E+00 -0.1443E-01
 7  0.7580E-03  0.0000E+00 -0.1435E-01
 8  0.1511E-02  0.0000E+00 -0.1411E-01
 9  0.0000E+00  0.0000E+00 -0.6164E-02
10  0.2792E-02  0.0000E+00 -0.6430E-02
.
.
.
66  0.0000E+00  0.1572E-02 -0.1028E-03
67 -0.7437E-04  0.1716E-02 -0.5845E-04
68  0.0000E+00  0.0000E+00  0.0000E+00
69  0.0000E+00  0.0000E+00  0.0000E+00
70  0.0000E+00  0.0000E+00  0.0000E+00

The integration point (nip= 1) stresses are:
Element      x-coord      y-coord      z-coord
sig_x        sig_y        sig_z        tau_xy      tau_yz      tau_zx
 1  0.2500E+00  0.5000E+00 -0.5000E+00
-0.2671E-01 -0.1647E+00 -0.9088E+00  0.6148E-02  0.9598E-01  0.4352E-02
 2  0.2500E+00  0.5000E+00 -0.1500E+01
0.3986E-01 -0.5316E-01 -0.6298E+00 -0.2139E-02  0.7613E-01  0.4169E-02
 3  0.2500E+00  0.1500E+01 -0.5000E+00
-0.2481E-01 -0.1260E+00 -0.1052E+00  0.4842E-02  0.9398E-01 -0.2812E-02
 4  0.2500E+00  0.1500E+01 -0.1500E+01
0.2477E-01 -0.8240E-01 -0.2822E+00 -0.3176E-02  0.1214E+00  0.2938E-02
 5  0.2500E+00  0.2500E+01 -0.5000E+00
0.3756E-02  0.8610E-02 -0.1470E-01 -0.9130E-03 -0.8730E-02  0.8741E-03
 6  0.2500E+00  0.2500E+01 -0.1500E+01
0.7401E-02 -0.4390E-01 -0.2832E-01 -0.5657E-03  0.6083E-01  0.4901E-04

```

**Figure 5.39** Results from Program 5.6 example

are used, namely sigma, dee, eps, ndi, nst, props, nprops, points, iel and npt. Swapping deemat for umat.f is straightforward and the program gives identical results. For further details about the variables used in the argument list and for guidance regarding how to develop user-written subroutines, the interested reader should consult the ABAQUS documentation. Example UMATs for different types of mechanical behaviour are provided by Dunne and Petrinic (2005).

```

*****
** UMATS FOR ABAQUS/STANDARD                                **
** ISOTROPIC ISOTHERMAL ELASTICITY                            **
** CANNOT BE USED FOR PLANE STRESS                           **
** PROPS(1) - E,    PROPS(2) - NU                            **
*****
*USER SUBROUTINE
    SUBROUTINE UMAT(STRESS,STATEV,DDSDDE,SSE,SPD,
1 SCD,RPL,DDSDDT,DRPLDE,DRPLDT,STRAN,DSTRAN,TIME,DTIME,
2 TEMP,DTEMP,PREDEF,DPRED,CMNAME,NDI,NSHR,NTENS,NSTATV,
3 PROPS,NPROPS,COORDS,DROT,PNEWDT,CELENT,DFGRD0,DFGRD1,
4 NOEL,NPT,LAYER,KSPT,KSTEP,KINC)

C     INCLUDE 'ABA_PARAM.INC'
C
CHARACTER*80 CMNAME
```

```

C
      DIMENSION STRESS(NTENS), STATEV(NSTATV), DROT(3,3),
1  DDSDDE(NTENS,NTENS), DDSDDT(NTENS), DRPLDE(NTENS),
2  STRAN(NTENS), DSTRAN(NTENS), TIME(2), PREDEF(1), DPRED(1),
3  PROPS(NPROPS), COORDS(3), DFGRD0(3,3), DFGRD1(3,3)
C
      PARAMETER (M=3,N=3,ID=3,ZERO=0.D0,ONE=1.D0,TWO=2.D0,
+ THREE=3.D0,SIX=6.D0, NINE=9.D0, TOLER=0.D-6)
C
      DIMENSION DSTRESS(4), DDS(4,4)
C
C ** TEST FOR NUMBER OF DIMENSIONS
C
      IF(NDI.NE.3) THEN
          WRITE(7,*) 'THIS UMAT MAY ONLY BE USED FOR ELEMENTS
1 WITH THREE DIRECT STRESS COMPONENTS'
          RETURN
      END IF
C
C ** ELASTIC PROPERTIES
C
      EMOD    = PROPS(1)
      ENU     = PROPS(2)
      EBULK3 = EMOD/(ONE-TWO*ENU)
      EG2    = EMOD/(ONE+ENU)
      EG     = EG2/TWO
      ELAM   = (EBULK3-EG2)/THREE
C
C ** ELASTIC STIFFNESS
C
C
      REAL ARRAY DDSDDE() IS THE "DEE" MATRIX IN S&G EDITION 4
C
      DO K1 = 1, NDI
          DO K2 = 1, NDI
              DDSDDE(K2,K1) = ELAM
          END DO
          DDSDDE(K1,K1) = EG2 + ELAM
      END DO
C
      DO K1 = NDI+1, NTENS
          DDSDDE(K1,K1) = EG
      END DO
C
C ** CALCULATE STRESS
C
C
      REAL ARRAY DSTRAN() IS "MATMUL(BEE,ELD)" IN THE BOOK
C
      THIS LOOP IS THE EQUIVALENT OF THE LINE:
C
      SIGMA=MATMUL(DEE,MATMUL(BEE,ELD))
C
      DO K1=1,NTENS
          DO K2=1,NTENS
              STRESS(K2)=STRESS(K2)+DDSDDE(K2,K1)*DSTRAN(K1)
          END DO
      END DO
C
      RETURN
END
**

```

```

nod nxe nye nze nip cg_tol cg_limit np_types
20    20    60    40    8   1.00E-05    2000      1

prop(E,v)
100.0  0.3

etype(not needed)

xcoords, ycoords, zcoords
0.0  0.025  0.050  0.075 ... 16 x-coords omitted...  0.5000
0.0  0.050  0.100  0.150 ... 56 y-coords omitted...  3.0000
0.0 -0.050 -0.100 -0.150 ... 36 z-coords omitted... -2.0000

nr,(k,nf(:,k),i=1,nr)
13441
 1 0 0 1       2 1 0 1       3 1 0 1       4 1 0 1       5 1 0 1
.
. nf data for 13431 nodes omitted
.
205437 0 0 0 205438 0 0 0 205439 0 0 0 205440 0 0 0 205441 0 0 0

loaded_nodes, (k,loads(nf(:,k)),i=1,loaded_nodes)
1281
 1  0.00000000E+00  0.00000000E+00  0.10416667E-03
 2  0.00000000E+00  0.00000000E+00 -0.41666667E-03
.
. loads data for 1277 nodes omitted
.
67680  0.00000000E+00  0.00000000E+00 -0.41666667E-03
67681  0.00000000E+00  0.00000000E+00  0.10416667E-03

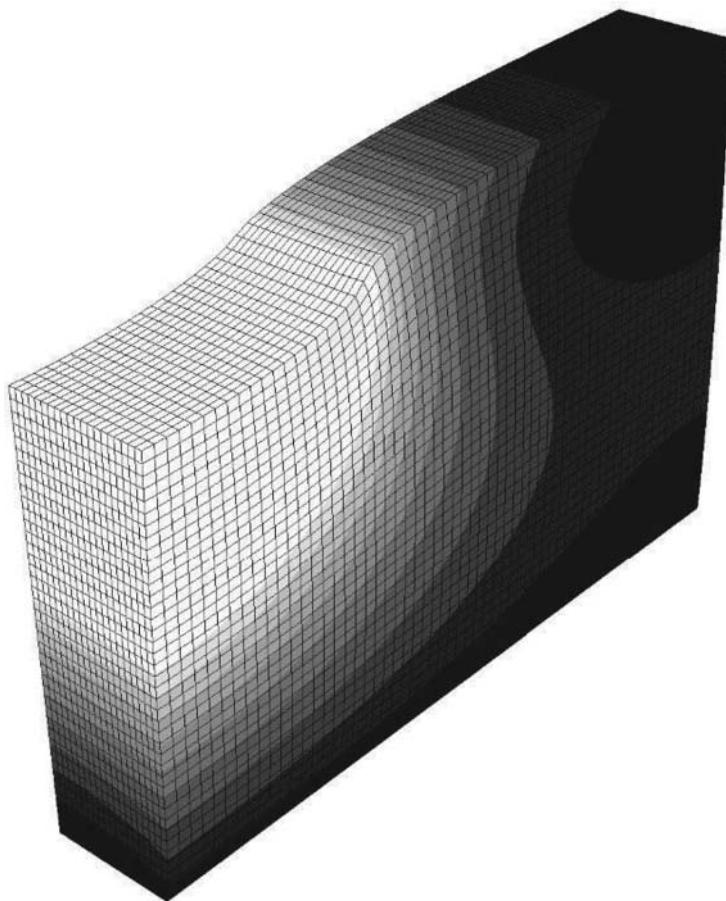
fixed_freedoms
0

```

**Figure 5.40** Data for Program 5.7 example

Nowadays, the architectures of commodity processors vary widely. To get the best performance out of the hardware, the use of the hardware vendor's own profiling tools and compilers is highly recommended. Profiling of Program 5.6 on a single core of an Intel i7 processor highlighted that most of the run time is spent in the pcg solver, particularly the `elements_3` loop, and is therefore a target for optimisation. A further set of changes to Program 5.6 concerns three ways in which the program run time can be reduced, as outlined below.

- (a) The optimisation that requires the least programming effort is to take advantage of the Intel i7 vector units (see Section 1.5). Free compilers such as g95 may not recognise hardware features specific to a particular release of a processor and here it is reasonably assumed that g95 gives us a baseline run time with vectorisation turned off. In contrast, the Intel compiler automatically analyses the source code during compilation and turns on vectorisation.
- (b) The second area for improvement in Program 5.6 is in the `elements_3` loop. Matrix–matrix computations can be carried out more efficiently than matrix–vector operations. If all the hexahedral elements have the same shape and material properties,



**Figure 5.41** Results from Program 5.7 example

which they do here, the element stiffness matrices are identical. It is therefore possible to replace the `elements_3` loop with a single matrix–matrix multiplication. Of course, this is a special case and real problems may comprise sets of identical elements and collections of unique ones.

- (c) A further reduction in run time comes from using the vendor's own highly optimised maths libraries. The FORTRAN intrinsic functions and subroutines may not be as highly tuned to the hardware as the vendor's own libraries. In Program 5.7, `MATMUL` has been replaced by `dsymm`, `DOT_PRODUCT` has been replaced by `ddot` and multiplying a vector by a scalar (e.g., `alpha*loads`) has been replaced by a call to the subroutine `daxpy`. See Appendix G for a description of external subprograms.

Table 5.1 shows the progressive effects of making changes to the coding in Program 5.6. The speed-up of Program 5.7 over Program 5.6 on this particular processor was by a factor of about 9, illustrating the benefit of using the vendor's tools, compilers and libraries if fast run times are important.

**Table 5.1** Timings for program optimisations

Original code (Program 5.6)	360 s
Matrix–vector vectorised	190 s
Matrix–matrix scalar	171 s
Matrix–matrix vectorised	83 s
Matrix–matrix vectorised + MKL BLAS (Program 5.7)	42 s

## 5.2 Glossary of Variable Names

### Scalar integers:

cg_iters	pcg iteration counter
cg_limit	pcg iteration ceiling
fixed_freedoms	number of fixed displacements
i_iel	simple counters
iflag	1 for ‘symmetry’, -1 for ‘antisymmetry’
iwp	SELECTED_REAL_KIND(15)
k	simple counter
kinc	unused ABAQUS UMAT variable
kspt	unused ABAQUS UMAT variable
kstep	unused ABAQUS UMAT variable
layer	unused ABAQUS UMAT variable
loaded_nodes	number of loaded nodes
lth	harmonic on which loads are to be applied
ndi	number of dimensions (ABAQUS UMAT)
ndim	number of dimensions
ndof	number of degrees of freedom per element
nels	number of elements
neq	number of degrees of freedom in the mesh
nip	number of integrating points per element
nlen	maximum number of characters in data file basename
nn	number of nodes in the mesh
nod	number of nodes per element
nodof	number of degrees of freedom per node
nprops	number of material properties
np_types	number of different property types
npt	unused ABAQUS UMAT variable
nr	number of restrained nodes
nre	number of elements in <i>r</i> -direction
nshr	unused ABAQUS UMAT variable
nspr	number of springs
nst	number of stress (strain) terms (3, 4 or 6)
nstatv	unused ABAQUS UMAT variable
nxe, nye, nze	number of elements in the <i>x</i> -, <i>y</i> - and <i>z</i> -directions

**Scalar reals:**

alpha,beta	$\alpha$ and $\beta$ from equations (3.22)
ca	set to $\cos(\chi)$
celent	unused ABAQUS UMAT variable
chi	angle for stress output
cg_tol	pcg convergence tolerance
det	determinant of the Jacobian matrix
dtime	unused ABAQUS UMAT variable
dtemp	unused ABAQUS UMAT variable
gtemp	Gauss point temperature change
one	set to 1.0
penalty	set to $1 \times 10^{20}$
pi	set to $\pi$
pnewdt	unused ABAQUS UMAT variable
radius	$r$ -coordinate of Gauss point
rpl	unused ABAQUS UMAT variable
sa	set to $\sin(\chi)$
scd	unused ABAQUS UMAT variable
spd	unused ABAQUS UMAT variable
sse	unused ABAQUS UMAT variable
temp	unused ABAQUS UMAT variable
up	holds dot product $(\mathbf{R}_k)^T (\mathbf{R}_k)$ from equations (3.22)
zero	set to 0.0

**Scalar characters:**

argv	holds data file basename
cmname	unused ABAQUS UMAT variable
dir	element and node numbering direction
element	element type
type_2d	type of 2D analysis ('plane' or 'axisymmetric')

**Scalar logical:**

cg_converged	set to .TRUE. if pcg process has converged
--------------	--

**Dynamic integer arrays:**

etype	element property type vector
g	element steering vector
g_g	global element steering vector
g_num	global element node numbers vector
kdiag	diagonal term location vector
nf	nodal freedom matrix
no	fixed freedom numbers vector
node	fixed nodes vector
num	element node numbers vector
sense	sense of freedoms to be fixed vector
sno	spring freedom number vector
spno	spring nodes vector
spse	sense of nodal spring vector

**Dynamic real arrays:**

bee	strain-displacement matrix
coord	element nodal coordinates
d	preconditioned rhs vector
ddssddt	unused ABAQUS UMAT variable
dee	stress-strain matrix
der	shape function derivatives with respect to local coordinates
deriv	shape function derivatives with respect to global coordinates
dfgrad0	unused ABAQUS UMAT variable
dfgrad1	unused ABAQUS UMAT variable
dpred	unused ABAQUS UMAT variable
drplde	unused ABAQUS UMAT variable
drpldt	unused ABAQUS UMAT variable
drot	unused ABAQUS UMAT variable
dtel	element nodal temperature changes vector
dtemp	global nodal temperature changes vector
diag_precon	diagonal preconditioner vector
eld	element nodal displacements
eps	strain terms
etl	element thermal forces vector
fun	shape functions
gc	integrating point coordinates
gravlo	loads generated by gravity
g_coord	global nodal coordinates
g_pmul	'gathers' the p vectors for all elements
g_utemp	holds all the products of km and p
jac	Jacobian matrix
km	element stiffness matrix
kv	global stiffness matrix
loads	nodal loads and displacements
p	'descent' vector used in equations (3.22)
points	integrating point local coordinates
predef	unused ABAQUS UMAT variable
prop	element properties
props	element properties (ABAQUS UMAT format)
r_coords	r-coordinates of mesh layout
sigma	stress terms
spva	spring stiffness vector
statev	unused ABAQUS UMAT variable
store	stores augmented diagonal terms
storkm	holds element stiffness matrices
stran	unused ABAQUS UMAT variable
teps	thermal strains
time	unused ABAQUS UMAT variable
tload	global thermal forces vector
u	vector used in equation (3.22)

<code>value</code>	fixed values of displacements
<code>weights</code>	weighting coefficients
<code>x, xnew</code>	'old' and 'new' solution vectors
<code>x_coords, y_coords, z_coords</code>	$x$ -, $y$ -, $z$ -coordinates of mesh layout

### 5.3 Exercises

- Derive in terms of local coordinates, any shape function of the following elements:
  - 6-node triangle
  - 8-node quadrilateral
  - 9-node quadrilateral

Answer: See Appendix B

- Given the following shape function of a 4-node plane stress rectangular element of width  $a$  and height  $b$

$$N_1 = \left(1 - \frac{x}{a}\right) \left(1 - \frac{y}{b}\right)$$

use analytical integration to show that the corresponding diagonal terms of the element stiffness and mass matrices are given by

$$k_{11} = \frac{E}{1-\nu^2} \left( \frac{b}{3a} + \frac{1-\nu}{2} \frac{a}{3b} \right)$$

$$m_{11} = \frac{\rho ab}{9}$$

- For the problem shown in Figure 5.42, estimate the force necessary to displace the loaded node horizontally by 0.015 units. Check your solution by using Program 5.1.

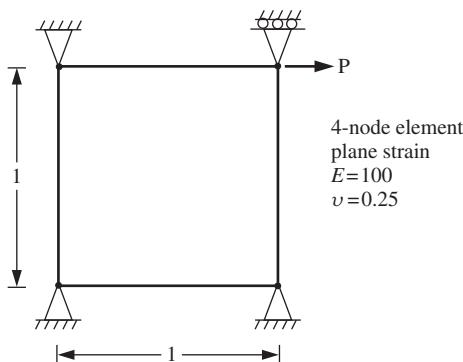


Figure 5.42

Answer: 0.8

4. Derive the vertical nodal forces that are equivalent to the triangular stress distribution acting on the 4-node plane-strain element shown in Figure 5.43. Given that the stiffness matrix of this element (assuming alternating  $u - v$  freedom numbering starting at the bottom left corner node and going clockwise) is

$$\begin{bmatrix} 57.69 & 24.04 & 9.62 & -4.81 & -28.85 & -24.04 & -38.46 & 4.81 \\ 57.69 & 4.81 & -38.46 & -24.04 & -28.85 & -4.81 & 9.62 & \\ 57.69 & -24.04 & -38.46 & -4.81 & -28.85 & 24.04 & & \\ 57.69 & 4.81 & 9.62 & 24.04 & -28.85 & & & \\ 57.69 & 24.04 & 9.62 & -4.81 & & & & \\ 57.69 & 4.81 & -38.46 & & & & & \\ & & 57.69 & -24.04 & & & & \\ & & & 57.69 & & & & \end{bmatrix}$$

compute the vertical displacement of the top two nodes. Check your solution using Program 5.1 given that  $E = 100$  and  $\nu = 0.3$ .

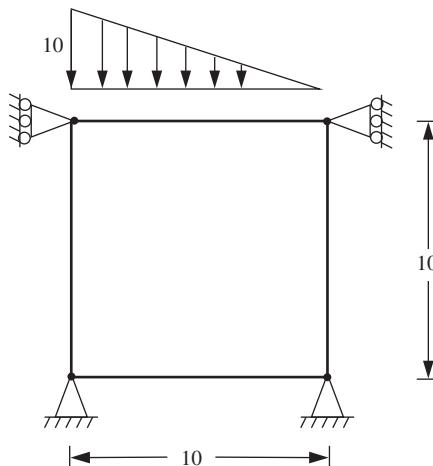


Figure 5.43

Answer:  $v_2 = -0.545$ ,  $v_3 = -0.198$

5. Derive the equivalent nodal force  $F_5$  required to model the linear distributed load shown in Figure 5.44. Given that  $F_4 = L/3$ , what must  $F_3$  be equal to?

Answer:  $F_3 = 0$ ,  $F_4 = L/3$ ,  $F_5 = L/6$

6. The 4-node element shown in Figure 5.45 is fixed on three sides and subjected to the indicated fixed displacement at the free node. Estimate the stresses at the centroid of the element. Check your solution using Program 5.1.

Answer:  $\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \begin{Bmatrix} 6.66 \\ 4.96 \\ -3.50 \end{Bmatrix}$

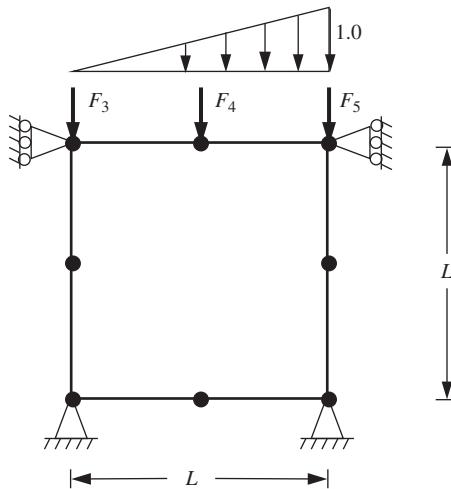


Figure 5.44

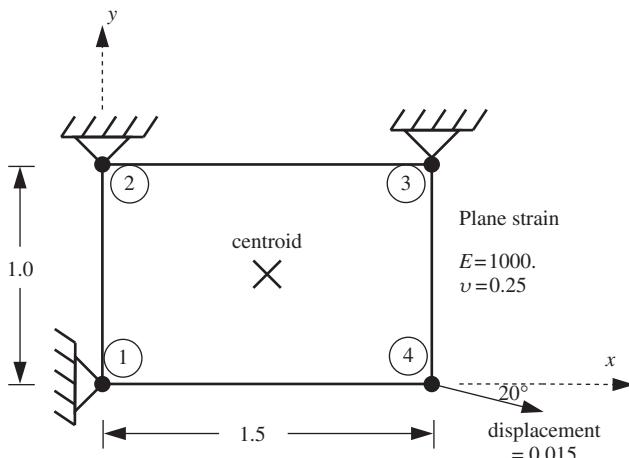


Figure 5.45

7. The rectangular 8-node element shown in Figure 5.46 is subjected to the gravitational body load  $F_y = -\gamma$ .

Compute the equivalent vertical nodal load  $F_{corner}$  at node 1.

If it can be assumed that the equivalent loads due to gravity result in all corner loads equalling  $F_{corner}$  and all mid-side loads equalling  $F_{mid-side}$ , deduce also the value of  $F_{mid-side}$ .

Answer:  $F_{corner} = ab\gamma/12$ ,  $F_{mid-side} = -ab\gamma/3$

8. The global stiffness matrix for the problem shown in Figure 5.47 is given by

$$[\mathbf{K}]_m = \begin{bmatrix} \alpha & 12.5 \\ 12.5 & 75 \end{bmatrix}$$

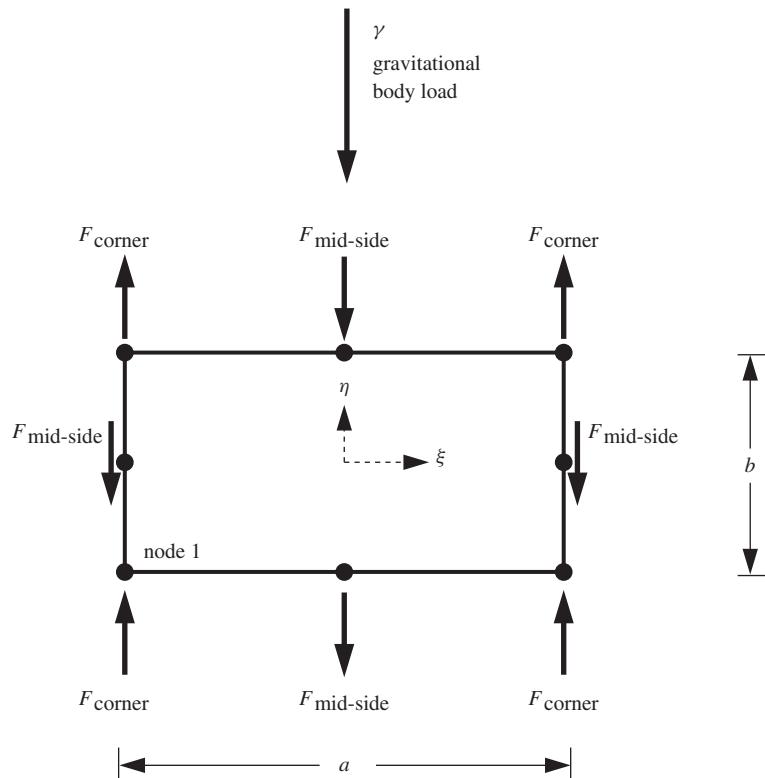


Figure 5.46

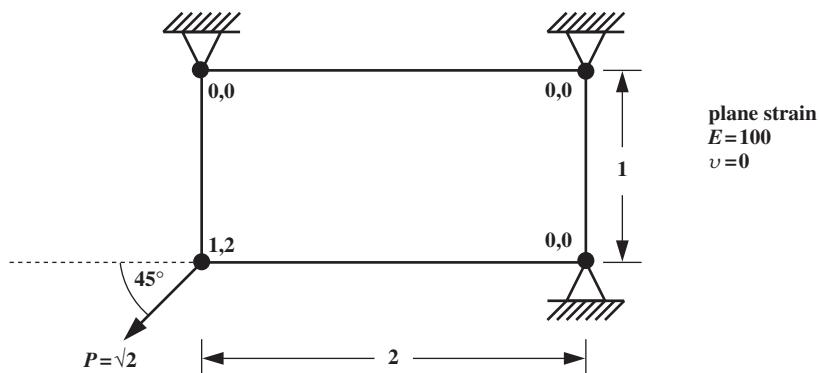


Figure 5.47

Derive the missing term  $\alpha$  and hence compute the displacement of the loaded node. Check your solution using Program 5.1.

Answer:  $\alpha = 50$ ,  $\delta_x = -0.017$ ,  $\delta_y = -0.010$

- Compute the equivalent nodal loads for the uniform distributed loading applied over one half of one side of the 8-node quadrilateral element shown in Figure 5.48.

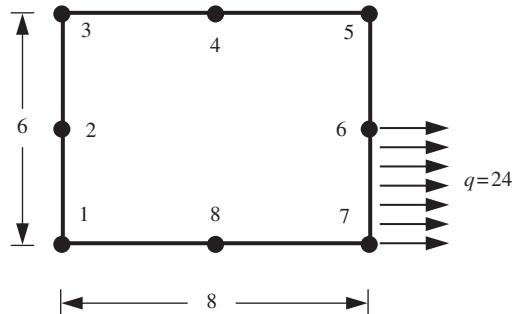


Figure 5.48

Answer:  $F_5 = -6$ ,  $F_6 = 48$ ,  $F_7 = 30$

10. The third member of the triangular family has 10 nodes as shown in Figure 5.49. Set up the system of simultaneous equations that would enable you to derive the shape function  $N_{10}$  in local coordinates for this element. Do not attempt to solve the equations.

Answer:  $N_{10} = c_1 + c_2 L_1 + c_3 L_2 + c_4 L_1^2 + c_5 L_1 L_2 + c_6 L_2^2 + c_7 L_1^3 + c_8 L_1^2 L_2 + c_9 L_1 L_2^2 + c_{10} L_2^3$

$$\begin{bmatrix} 27 & 27 & 0 & 27 & 0 & 0 & 27 & 0 & 0 & 0 \\ 27 & 0 & 27 & 0 & 0 & 27 & 0 & 0 & 0 & 27 \\ 27 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 27 & 18 & 9 & 12 & 6 & 3 & 8 & 4 & 2 & 1 \\ 27 & 9 & 18 & 3 & 6 & 12 & 1 & 2 & 4 & 8 \\ 27 & 0 & 18 & 0 & 0 & 12 & 0 & 0 & 0 & 8 \\ 27 & 0 & 9 & 0 & 0 & 3 & 0 & 0 & 0 & 1 \\ 27 & 9 & 0 & 3 & 0 & 0 & 1 & 0 & 0 & 0 \\ 27 & 18 & 0 & 12 & 0 & 0 & 8 & 0 & 0 & 0 \\ 27 & 9 & 9 & 3 & 3 & 3 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \\ c_{10} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 27 \end{Bmatrix}$$

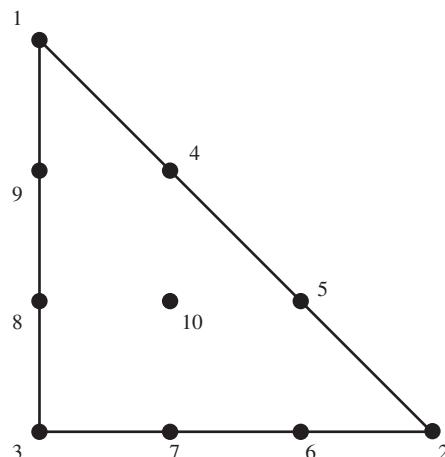
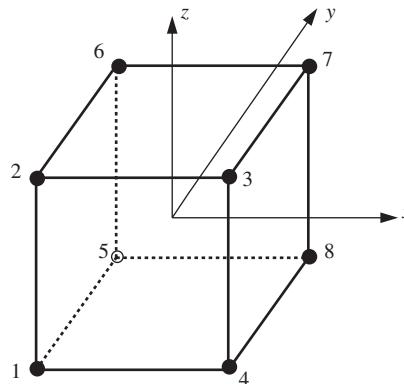


Figure 5.49

11. Figure 5.50 shows a cubic, 8-node, 3D finite element with side length 2 units. The origin of the coordinate system is at the centroid of the element, so all nodal coordinates are  $\pm 1$  [e.g., node 1 is at  $(-1, -1, -1)$ , etc.]. Choose suitable terms for the shape functions of this element, and hence set up the system of simultaneous equations that would enable you to derive shape function  $N_7$ . Do not attempt to solve the equations.



**Figure 5.50**

Answer:  $N_7 = c_1 + c_2x + c_3y + c_4z + c_5xy + c_6yz + c_7zx + c_8xyz$

$$\begin{bmatrix} 1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & 1 & -1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & 1 & 1 & -1 & 1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & -1 & 1 & -1 & -1 & -1 \end{bmatrix} \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{Bmatrix}$$

12. Assuming ‘small’ strains, derive the partial differential equations of 2D elastic equilibrium under conditions of plane stress as given in (2.57).
13. Following the previous question, show that the corresponding partial differential equations under plane-strain conditions are given by

$$\frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \left\{ \begin{array}{l} \frac{\partial^2 u}{\partial x^2} + \frac{1-2\nu}{2(1-\nu)} \frac{\partial^2 u}{\partial y^2} + \frac{\nu}{1-\nu} \frac{\partial^2 v}{\partial x \partial y} + \frac{1-2\nu}{2(1-\nu)} \frac{\partial^2 v}{\partial y \partial x} \\ \frac{\partial^2 v}{\partial y^2} + \frac{1-2\nu}{2(1-\nu)} \frac{\partial^2 v}{\partial x^2} + \frac{\nu}{1-\nu} \frac{\partial^2 u}{\partial y \partial x} + \frac{1-2\nu}{2(1-\nu)} \frac{\partial^2 u}{\partial x \partial y} \end{array} \right\} = \begin{Bmatrix} -F_x \\ -F_y \end{Bmatrix}$$

14. Using the shape functions of (2.60), compute the term  $k_{12}$  of the square element shown in Figure 5.51.

Answer:  $k_{12} = E\nu/6(1-\nu^2)$

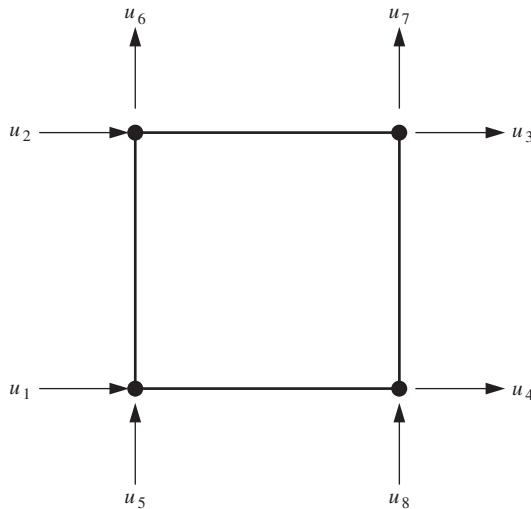


Figure 5.51

15. Selective reduced integration (SRI) is a way of improving the performance of 4-node plane elements as Poisson's ratio approaches 0.5 (incompressibility). The  $[\mathbf{D}]$  matrix is split into volumetric and deviatoric components, and the element stiffness matrix is integrated in two stages, namely

$$[\mathbf{k}_m] = \int \int [\mathbf{B}]^T [\mathbf{D}]^v [\mathbf{B}] dx dy + \int \int [\mathbf{B}]^T [\mathbf{D}]^d [\mathbf{B}] dx dy$$

where

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

and

$$[\mathbf{D}]^v = \frac{Ev}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

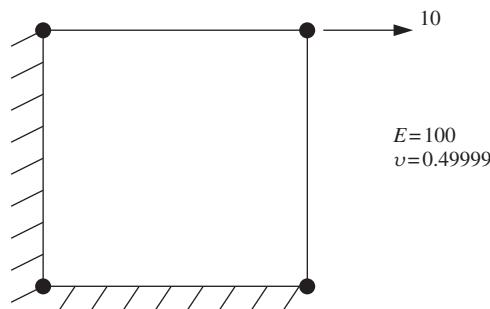
The SRI approach involves 'exact' integration ( $nip = 4$ ) of the  $[\mathbf{D}]^d$  term, and 'reduced' integration ( $nip = 1$ ) of the  $[\mathbf{D}]^v$  term.

Use this technique to estimate the displacement of the loaded node of the square element of side length 2 in Figure 5.52. You may use analytical integration for the  $[\mathbf{D}]^d$  term.

Answer:  $\delta_x = 0.2$ ,  $\delta_x = -0.2$

16. A planar square 8-node quadrilateral of unit side length and unit mass density has the following shape functions at node 1:

$$N_1 = 0.25(1-\xi)(1-\eta)(-\eta-\xi-1)$$

**Figure 5.52**

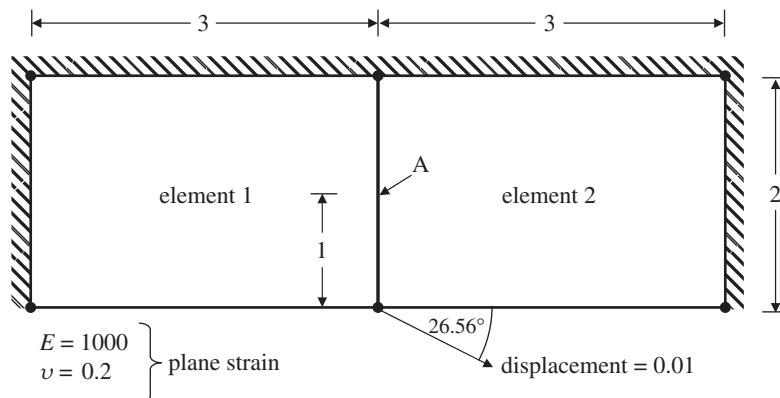
Compute  $m_{11}$  of the element consistent mass matrix using (a)  $nip=4$  and (b)  $nip=9$ .

Answer: (a)  $m_{11} = 0.0185$ , (b)  $m_{11} = 0.0333$

17. Given the elastic material shown in Figure 5.53, compute the stresses at point A based on:

- (a) the deformation of element 1 and
- (b) the deformation of element 2.

What do your results suggest about the stress equilibrium between adjacent elements?

**Figure 5.53**

Answer:

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \begin{Bmatrix} 2.277 \\ 2.899 \\ -2.174 \end{Bmatrix}_{A1} \quad \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \begin{Bmatrix} -1.036 \\ 2.070 \\ -1.553 \end{Bmatrix}_{A2}$$

18. Use Program 5.1 with 8-node square elements of side length 0.2 throughout to compute the vertical deflection at the centre and edge of the flexible footing shown in Figure 5.54. You should use symmetry to reduce the number of elements in the mesh.

Answer: 0.0905, 0.0542

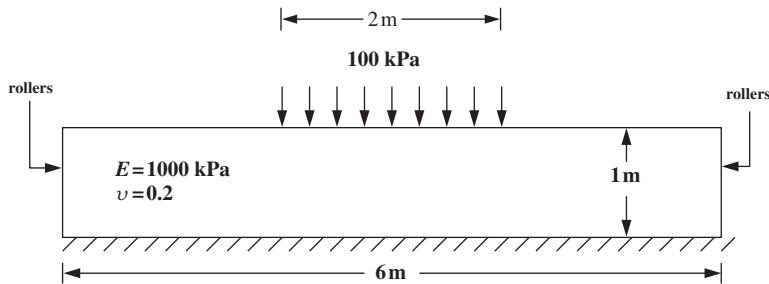


Figure 5.54

## References

- Cuthill E and McKee J 1969 Reducing the bandwidth of sparse symmetric matrices. *ACM Proceedings of the 24th National Conference*, ACM, New York.
- Dunne F and Petrinic N 2005 *Introduction to Computational Plasticity*. Oxford University Press, Oxford.
- Griffiths DV 1986 HARMONY—A program for predicting the response of axisymmetric bodies subjected to non-axisymmetric loading. Technical Report GRC-96-44, Arthur Lakes Library, Colorado School of Mines, Geomechanics Research Center.
- Hicks MA and Mar A 1996 A benchmark computational study of finite element error estimation. *Int J Numer Methods Eng* **39**(23), 3969–3983.
- Irons BM 1971 Quadrature rules for brick-based finite elements. *Int J Numer Methods Eng* **3**, 293–294.
- Johns DJ 1965 *Thermal Stress Analyses*. Pergamon, New York.
- Peano A 1987 Inadmissible distortion of solid elements and patch tests results. *Comm Appl Numer Methods* **5**, 97–101.
- Poulos HG and Davis EH 1974 *Elastic Solutions for Soil and Rock Mechanics*. John Wiley & Sons, Chichester.
- Smith IM and Kidger DJ 1992 Elastoplastic analysis using the 14-node brick element family. *Int J Numer Methods Eng* **35**, 1263–1275.
- Wilson EL 1965 Structural analysis of axisymmetric solids. *J Am Inst Aeronaut Astronaut* **3**, 2269–2274.
- Zienkiewicz OC, Taylor RL and Zhu JZ 2005 *The Finite Element Method*, Vol. 1, 6th edn. McGraw-Hill, London.

# 6

# Material Non-linearity

## 6.1 Introduction

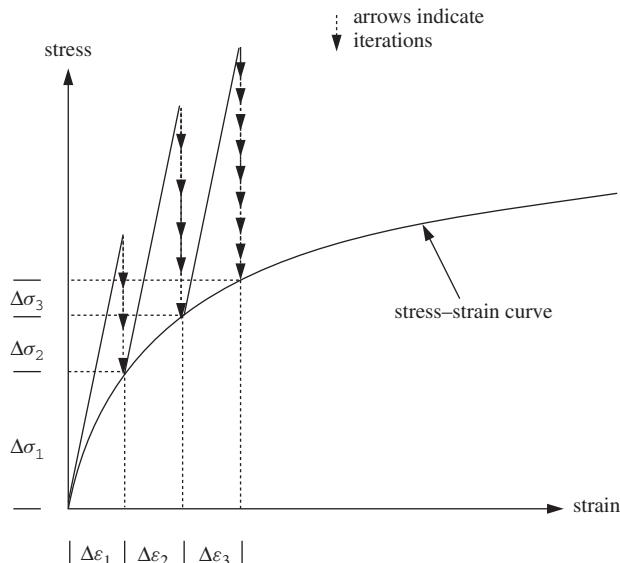
Non-linear processes pose very much greater analytical problems than do the linear processes considered for the most part so far in this book. The non-linearity may be found in the dependence of the equation coefficients on the solution itself or in the appearance of powers and products of the unknowns or their derivatives.

Two main types of non-linearity can manifest themselves in finite element analysis of solids: material non-linearity, in which the relationship between stresses and strains (or other material properties) are complicated functions which result in the equation coefficients depending on the solution, and geometric non-linearity (otherwise known as ‘large-strain’ or ‘large-displacement’ analysis), which leads to products of the unknowns in the equations.

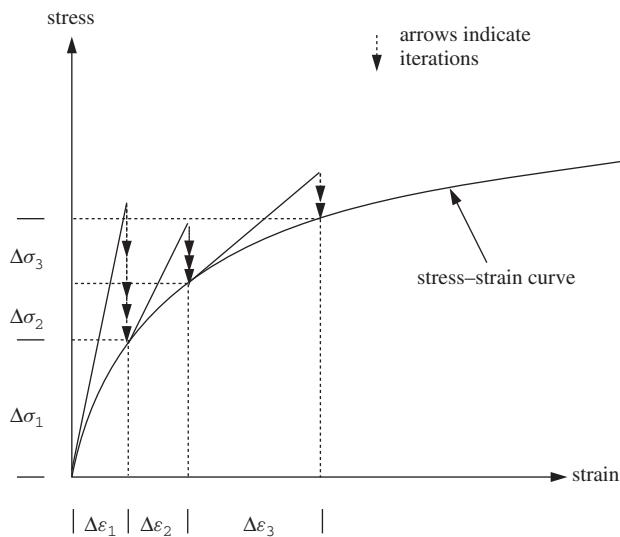
In order to keep the present book to a manageable size, the 13 programs described in this chapter deal only with material non-linearity. As far as the organisation of computer programs is concerned, material non-linearity is simpler to implement than geometric non-linearity. However, readers will appreciate how programs could be adapted to cope with geometric non-linearity as well (see, e.g., Smith, 1997).

In practical finite element analysis two main types of solution procedure can be adopted to model material non-linearity. The first approach, that has already been seen in Program 4.5, involves ‘constant stiffness’ iterations in which non-linearity is introduced by iteratively modifying the right-hand-side ‘loads’ vector. The (usually elastic) global stiffness matrix in such an analysis is formed once only. Each iteration thus represents an elastic analysis of the type described in Chapter 5. Convergence is said to occur when stresses generated by the loads satisfy some stress–strain law or yield or failure criterion within prescribed tolerances. The loads vector at each iteration consists of externally applied loads and self-equilibrating ‘body loads’. The body loads have the effect of redistributing stresses (or moments) within the system, but as they are self-equilibrating, they do not alter the net loading on the system. The ‘constant stiffness’ method is shown diagrammatically in Figure 6.1. For load-controlled problems, many iterations may be required as failure is approached, because the elastic (constant) global stiffness matrix starts to seriously overestimate the actual material stiffness.

Fewer iterations per load step are required if the second approach, the ‘variable’ or ‘tangent’ stiffness method is adopted. This method, shown in Figure 6.2, takes account



**Figure 6.1** Constant stiffness method



**Figure 6.2** Variable (tangent) stiffness method

of the reduction in stiffness of the material as failure is approached. If small enough load steps are taken, the method can become equivalent to a simple Euler ‘explicit’ method. In practice, the global stiffness matrix may be updated periodically and ‘body loads’ iterations employed to achieve convergence. In contrasting the two methods, the extra cost of reforming and factorising the global stiffness matrix in the ‘variable stiffness’ method is offset by reduced numbers of iterations, especially as failure is approached.

A further possibility, introduced in later programs, is ‘implicit’ integration of the rate equations rather than the ‘explicit’ methods just described. This helps to further reduce the number of iterations to convergence.

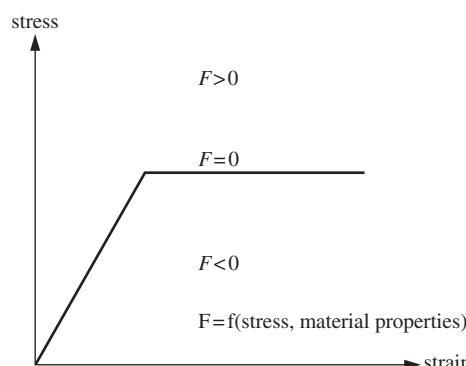
Programs 6.1 to 6.5 employ the constant stiffness algorithms and are similar in structure to Program 4.5 described previously for plastic analysis of frames. Both the viscoplastic method and the simple initial stress method are implemented. Programs 6.6 to 6.8 introduce variable stiffness algorithms, and Programs 6.9 and 6.10 describe procedures for embanking and excavation in which construction sequence can be more realistically modelled. Program 6.11 introduces a simple technique for modelling pore pressures in a saturated soil, and Programs 6.12 and 6.13 complete the chapter with 3D elastic–plastic analyses of slopes. ‘Element-by-element’ solution strategies are described in Programs 6.2, 6.7 and 6.13. Many of the examples are chosen because they have alternative solutions that can be used for comparison.

Before describing the programs, some discussion is necessary regarding the form of the stress–strain laws that are to be adopted. In addition, two popular methods of generating body loads for ‘constant stiffness’ methods, namely ‘viscoplasticity’ and ‘initial stress’ are described.

## 6.2 Stress–strain Behaviour

Although non-linear elastic constitutive relations have been applied in finite element analyses and especially soil mechanics applications (e.g., Duncan and Chang, 1970), the main physical feature of non-linear material behaviour is usually the irrecoverability of strain. A convenient mathematical framework for describing this phenomenon is to be found in the theory of plasticity (e.g., Hill, 1950). The simplest stress–strain law of this type that could be implemented in a finite element analysis involves elastic–perfectly plastic material behaviour as shown in Figure 6.3.

Although a simple law of this type was described in Chapter 4 (Figure 4.28), it is convenient in solid mechanics to introduce a ‘yield’ surface in principal stress space which separates stress states that give rise to elastic and to plastic (irrecoverable) strains. To take account of complicated processes like cyclic loading, the yield surface may move in stress space ‘kinematically’ (e.g., Molenkamp, 1987), but in this book only immovable surfaces are considered. An additional simplification introduced here is that the yield and ultimate ‘failure’ surfaces are identical.



**Figure 6.3** Elastic–perfectly plastic stress–strain law

Algebraically, the surfaces are expressed in terms of a yield or failure function  $F$ . This function, which has units of stress, depends on the material strength and invariant combinations of the stress components. The function is designed such that it is negative within the yield or failure surface and zero on the yield or failure surface. Positive values of  $F$  imply stresses lying outside the yield or failure surface which are illegal and which must be redistributed via the iterative process described previously.

During plastic straining, the material may flow in an ‘associated’ manner, that is the vector of plastic strain increment may be normal to the yield or failure surface. Alternatively, normality may not exist and the flow may be ‘non-associated’. Associated flow leads to various mathematically attractive simplifications and, when allied to the von Mises or Tresca failure criterion, correctly predicts zero plastic volume change during yield for undrained clays. For frictional materials, whose ultimate state may be described by the Mohr–Coulomb criterion, associated flow leads to physically unrealistic volumetric expansion or dilation during yield. In such cases, non-associated flow rules are preferred in which plastic straining is described by a plastic potential function  $Q$ . This function may be geometrically similar to the failure function  $F$  but with the friction angle  $\phi$  replaced by a dilation angle  $\psi$ . The implementation of the plastic potential function will be described further in Sections 6.6 and 6.7.

Before outlining some commonly used failure criteria and their representations in principal stress space, some useful stress invariant expressions are reviewed briefly.

### 6.3 Stress Invariants

The Cartesian stress tensor defining the stress conditions at a point within a loaded body is given by

$$[\sigma_x \ \sigma_y \ \sigma_z \ \tau_{xy} \ \tau_{yz} \ \tau_{zx}]^T \quad (6.1)$$

which can be shown to be equivalent to three principal stresses acting on orthogonal planes:

$$[\sigma_1 \ \sigma_2 \ \sigma_3]^T \quad (6.2)$$

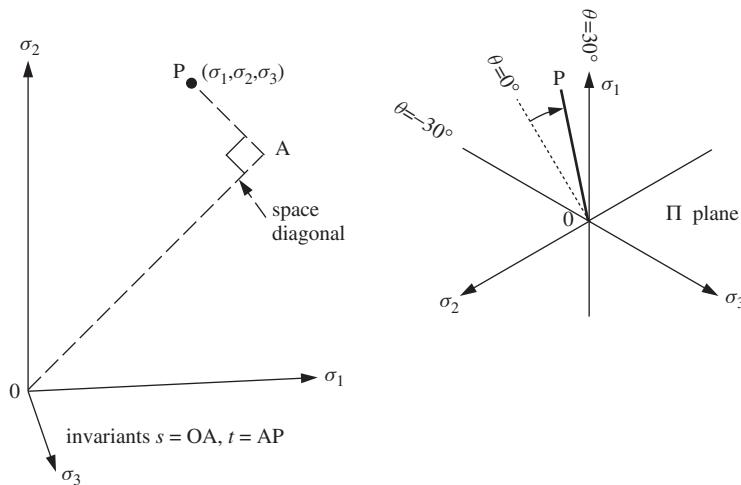
Principal stress space is obtained by treating the principal stresses as three-dimensional coordinates and is a useful way of representing a stress state at a point. It may be noted that although principal stress space defines the magnitudes of the principal stresses, it gives no indication of their orientation in physical space.

Instead of defining a point in principal stress space with coordinates  $(\sigma_1, \sigma_2, \sigma_3)$ , it is often more convenient to use invariants  $(s, t, \theta)$  defined as

$$\begin{aligned} s &= \frac{1}{\sqrt{3}}(\sigma_x + \sigma_y + \sigma_z) \\ t &= \frac{1}{\sqrt{3}}[(\sigma_x - \sigma_y)^2 + (\sigma_y - \sigma_z)^2 + (\sigma_z - \sigma_x)^2 + 6\tau_{xy}^2 + 6\tau_{yz}^2 + 6\tau_{zx}^2]^{1/2} \\ \theta &= \frac{1}{3} \arcsin \left( \frac{-3\sqrt{6}J_3}{t^3} \right) \end{aligned} \quad (6.3)$$

where

$$J_3 = s_x s_y s_z - s_x \tau_{yx}^2 - s_z \tau_{zx}^2 - s_z \tau_{xy}^2 + 2\tau_{xy} \tau_{yz} \tau_{zx}$$



**Figure 6.4** Representation of stress in principal stress space

and

$$s_x = \frac{(2\sigma_x - \sigma_y - \sigma_z)}{3} \quad \text{etc.}$$

As shown in Figure 6.4,  $s$  gives the distance from the origin to the ‘ $\pi$ -plane’ in which the stress point lies, and  $t$  represents the perpendicular distance of the stress point from the space diagonal. The Lode angle  $\theta$  (called `lode_theta` in the programs) is a measure of the angular position of the stress point within the  $\pi$ -plane.

It may be noted that in some geotechnical applications, plane-strain conditions apply and equations (6.3) are simplified because  $\tau_{yz} = \tau_{zx} = 0$ .

In the programs described later in this chapter, the invariants that are used are slightly different from those defined in (6.3), whence

$$\begin{aligned}\sigma_m &= \frac{1}{\sqrt{3}} s \\ \bar{\sigma} &= \sqrt{\frac{3}{2}} t\end{aligned}\tag{6.4}$$

These expressions, called `sigm` and `dsbar` in program terminology, have more physical meaning than  $s$  and  $t$  in that they represent respectively the ‘mean stress’ and ‘deviator stress’ in a triaxial test. The relationship between principal stresses and invariants is given as follows:

$$\begin{aligned}\sigma_1 &= \sigma_m + \frac{2}{3} \bar{\sigma} \sin\left(\theta - \frac{2\pi}{3}\right) \\ \sigma_2 &= \sigma_m + \frac{2}{3} \bar{\sigma} \sin\theta \\ \sigma_3 &= \sigma_m + \frac{2}{3} \bar{\sigma} \sin\left(\theta + \frac{2\pi}{3}\right)\end{aligned}\tag{6.5}$$

which ensures that  $\sigma_1$  is the most compressive and  $\sigma_3$  is the least compressive. Assuming compression is negative, the Lode angle  $\theta$  varies in the range  $-30^\circ \leq \theta \leq 30^\circ$ , where  $\theta = 30^\circ$  corresponds to ‘triaxial compression’ ( $\sigma_1 \leq \sigma_2 = \sigma_3$ ), and  $\theta = -30^\circ$  corresponds to ‘triaxial extension’ ( $\sigma_1 = \sigma_2 \leq \sigma_3$ ).

## 6.4 Failure Criteria

Several failure criteria have been proposed for representing the strength of soils as engineering materials. For soils with both frictional and cohesive components of shear strength, conical failure criteria are appropriate, the best known of which is undoubtedly the Mohr–Coulomb criterion. For metals or undrained clays which behave in a ‘frictionless’ ( $\phi_u = 0$ ) manner, cylindrical failure criteria are appropriate and are discussed first.

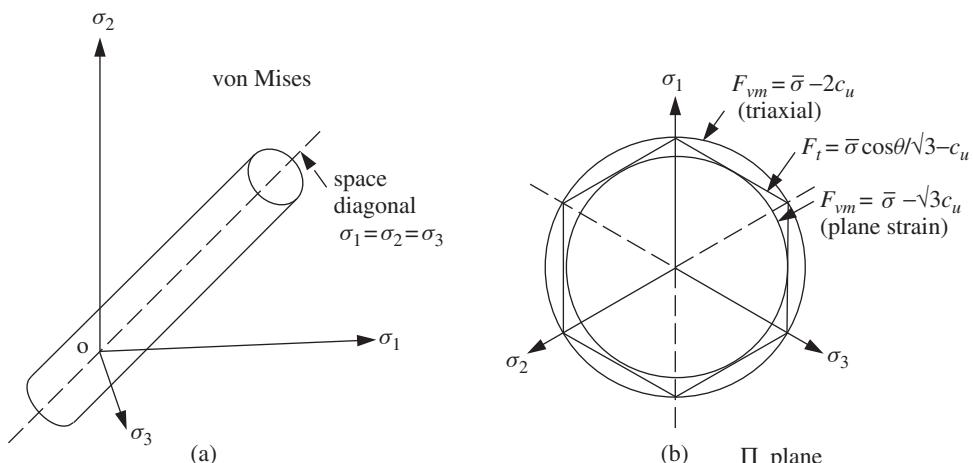
### 6.4.1 Von Mises

As shown in Figure 6.5(a), this criterion takes the form of a right-circular cylinder lying along the space diagonal. Only one of the three invariants, namely  $t$  (or  $\bar{\sigma}$ ), is of any significance when determining whether a stress state has reached the limit of elastic behaviour. The onset of yield in a von Mises material is not dependent upon invariants  $s$  or  $\theta$ .

The symmetry of the von Mises criterion when viewed in the  $\pi$ -plane indicates why it is not ideally suited to correlations with traditional soil mechanics concepts of strength. The criterion gives equal weighting to all three principal stresses, so if it is to be used to model undrained clay behaviour, consideration must be given to the value of the intermediate principal stress,  $\sigma_2$ , at failure.

For plane-strain applications assuming no plastic volume change, it can be shown that at failure

$$\sigma_2 = \frac{\sigma_1 + \sigma_3}{2} \quad (6.6)$$



**Figure 6.5** von Mises and Tresca failure criteria

Hence the von Mises criterion given by

$$F_{vm} = \bar{\sigma} - \sqrt{3}c_u \quad (6.7)$$

should be used, where  $c_u$  is the undrained ‘cohesion’ or shear strength of the soil.

In contrast, under triaxial conditions, where

$$\sigma_2 = \sigma_3 \quad (6.8)$$

the required von Mises criterion is given by

$$F_{vm} = \bar{\sigma} - 2c_u \quad (6.9)$$

Both of these expressions, when applied to the appropriate stress condition, ensure that at failure

$$\left| \frac{\sigma_1 - \sigma_3}{2} \right| = c_u \quad (6.10)$$

#### 6.4.2 Mohr–Coulomb and Tresca

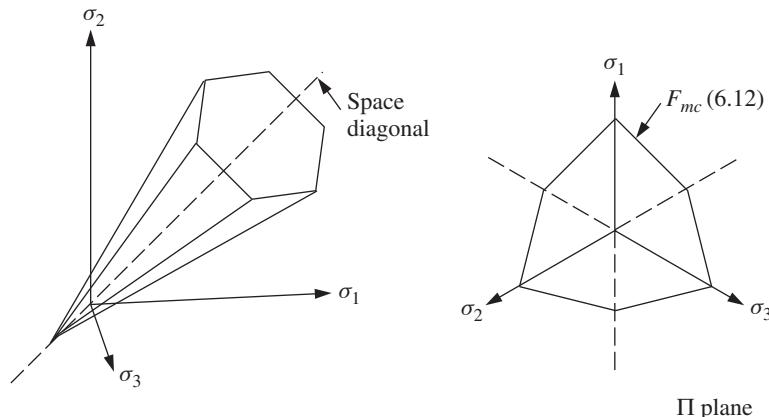
In principal stress space, the Mohr–Coulomb criterion takes the form of an irregular hexagonal cone, as shown in Figure 6.6. The irregularity is due to the fact that  $\sigma_2$  is not taken into account. In order to derive the invariant form of this criterion, it should first be written in terms of principal stresses from the geometry of Mohr’s circle, thus (assuming compression-negative):

$$F_{mc} = \frac{\sigma_1 + \sigma_3}{2} \sin \phi - \frac{\sigma_1 - \sigma_3}{2} - c \cos \phi \quad (6.11)$$

Substituting for  $\sigma_1$  and  $\sigma_3$  from (6.5) gives the function

$$F_{mc} = \sigma_m \sin \phi + \bar{\sigma} \left( \frac{\cos \theta}{\sqrt{3}} - \frac{\sin \theta \sin \phi}{3} \right) - c \cos \phi \quad (6.12)$$

which shows that the Mohr–Coulomb criterion depends on all three invariants ( $\sigma_m$ ,  $\bar{\sigma}$ ,  $\theta$ ).



**Figure 6.6** Mohr–Coulomb failure criterion

The Tresca criterion is obtained from (6.12) by putting  $\phi = 0$  to give

$$F_t = \frac{\bar{\sigma} \cos \theta}{\sqrt{3}} - c_u \quad (6.13)$$

This criterion is preferred to von Mises for applications involving undrained clays because (6.10) is always satisfied at failure, regardless of the value of  $\sigma_2$ . In principal stress space the Tresca criterion is a regular hexagonal cylinder lying in between the two versions of von Mises given by (6.7) and (6.9), as shown in Figure 6.5(b).

## 6.5 Generation of Body Loads

Constant stiffness methods of the type described in this chapter use repeated elastic solutions to achieve convergence by iteratively varying the loads on the system. Within each load increment, the system of equations

$$[\mathbf{K}_m]\{\mathbf{U}\}^i = \{\mathbf{F}\}^i \quad (6.14)$$

must be solved for the global displacement increments  $\{\mathbf{U}\}^i$ , where  $i$  represents the iteration number,  $[\mathbf{K}_m]$  the global stiffness matrix and  $\{\mathbf{F}\}^i$  the global external and internal (body) loads.

The element displacement increments  $\{\mathbf{u}\}^i$  are extracted from  $\{\mathbf{U}\}^i$ , and these lead to strain increments via the element strain–displacement relationships

$$\{\Delta\epsilon\}^i = [\mathbf{B}]\{\mathbf{u}\}^i \quad (6.15)$$

Assuming the material is yielding, the strains will contain both elastic and plastic components, thus

$$\{\Delta\epsilon\}^i = \{\Delta\epsilon^e\}^i + \{\Delta\epsilon^p\}^i \quad (6.16)$$

It is only the elastic strain increments  $\{\Delta\epsilon^e\}^i$  that generate stresses through the elastic stress–strain matrix, hence

$$\{\Delta\sigma\}^i = [\mathbf{D}^e]\{\Delta\epsilon^e\}^i \quad (6.17)$$

These stress increments are added to stresses already existing from the previous load step and the updated stresses substituted into the failure criterion [e.g., (6.12)]. If stress redistribution is necessary ( $F > 0$ ), this is done by altering the load increment vector  $\{\mathbf{F}\}^i$  in equation (6.14). In general, this vector holds two types of load, as given by

$$\{\mathbf{F}\}^i = \{\mathbf{F}_a\} + \{\mathbf{F}_b\}^i \quad (6.18)$$

where  $\{\mathbf{F}_a\}$  is the actual applied external load increment and  $\{\mathbf{F}_b\}^i$  is the body-loads vector that varies from one iteration to the next. The  $\{\mathbf{F}_b\}^i$  vector must be self-equilibrating so that the net loading on the system is not affected by it. Two simple methods for generating body loads are now described briefly.

## 6.6 Viscoplasticity

In this method (Zienkiewicz and Cormeau, 1974), the material is allowed to sustain temporarily, stresses outside the failure criterion. Overshoot of the failure criterion, as signified by  $F > 0$ , is an integral part of the method and is used to drive the algorithm.

Instead of plastic strains, we now refer to viscoplastic strains and these are generated at a rate that is related to the amount by which yield has been violated through the expression

$$\{\dot{\epsilon}^{vp}\} = F \left\{ \frac{\partial Q}{\partial \sigma} \right\} \quad (6.19)$$

where  $F$  is the yield function and  $Q$  is the plastic potential function.

It should be noted that a pseudo-viscosity property equal to unity is implied on the right-hand side of equation (6.19) from dimensional considerations.

Multiplication of the viscoplastic strain rate by a pseudo-time step gives an increment of viscoplastic strain which is accumulated from one ‘time step’ or iteration to the next; thus

$$\{\delta \epsilon^{vp}\}^i = \Delta t \{\dot{\epsilon}^{vp}\}^i \quad (6.20)$$

and

$$\{\Delta \epsilon^{vp}\}^i = \{\Delta \epsilon^{vp}\}^{i-1} + \{\delta \epsilon^{vp}\}^i \quad (6.21)$$

The ‘time step’ for unconditional numerical stability has been derived by Cormeau (1975) and depends on the assumed failure criterion. Thus, for von Mises materials:

$$\Delta t = \frac{4(1+\nu)}{3E} \quad (6.22)$$

and for Mohr–Coulomb materials:

$$\Delta t = \frac{4(1+\nu)(1-2\nu)}{E(1-2\nu + \sin^2 \phi)} \quad (6.23)$$

The derivatives of the plastic potential function  $Q$  with respect to stresses, as needed by (6.19), are conveniently expressed through the chain rule thus

$$\left\{ \frac{\partial Q}{\partial \sigma} \right\} = \frac{\partial Q}{\partial \sigma_m} \left\{ \frac{\partial \sigma_m}{\partial \sigma} \right\} + \frac{\partial Q}{\partial J_2} \left\{ \frac{\partial J_2}{\partial \sigma} \right\} + \frac{\partial Q}{\partial J_3} \left\{ \frac{\partial J_3}{\partial \sigma} \right\} \quad (6.24)$$

where  $J_2 = t^2/2$ . The viscoplastic strain rate is then evaluated numerically by an expression of the form

$$\{\dot{\epsilon}^{vp}\} = F \left( \frac{\partial Q}{\partial \sigma_m} [\mathbf{M}^1] + \frac{\partial Q}{\partial J_2} [\mathbf{M}^2] + \frac{\partial Q}{\partial J_3} [\mathbf{M}^3] \right) \{\sigma\} \quad (6.25)$$

where  $\partial Q/\partial \sigma_m$ ,  $\partial Q/\partial J_2$  and  $\partial Q/\partial J_3$  are represented by variables  $dq1$ ,  $dq2$  and  $dq3$  in the computer programs, and  $\{\partial \sigma_m/\partial \sigma\}$ ,  $\{\partial J_2/\partial \sigma\}$  and  $\{\partial J_3/\partial \sigma\}$  by the matrix–vector products  $[\mathbf{M}^1]\{\sigma\}$ ,  $[\mathbf{M}^2]\{\sigma\}$  and  $[\mathbf{M}^3]\{\sigma\}$ . This is essentially the notation used by Zienkiewicz *et al.* (2005), and these quantities are given in more detail in Appendix C.

The body loads  $\{\mathbf{F}_b\}^i$  are accumulated at each ‘time step’ within each load step by summing the following integrals for all elements containing a yielding ( $F > 0$ ) Gauss point:

$$\{\mathbf{F}_b\}^i = \{\mathbf{F}_b\}^{i-1} + \sum_{elements}^{all} \int \int [\mathbf{B}]^T [\mathbf{D}^e] \{\delta \epsilon^{vp}\}^i dx dy \quad (6.26)$$

This process is repeated at each ‘time step’ until no integrating point stresses violate the failure criterion to within a certain tolerance. The convergence criterion is based on a dimensionless measure of the amount by which the displacement increment vector  $\{\mathbf{U}\}^i$  changes from one iteration to the next. The convergence checking process is identical to that used in Program 4.5.

## 6.7 Initial Stress

The viscoplastic algorithm is often referred to as an ‘initial strain’ method to distinguish it from the more widely used ‘initial stress’ approaches (e.g., Zienkiewicz *et al.*, 1969).

Initial stress methods involve an explicit relationship between increments of stress and increments of strain. Thus, whereas linear elasticity was described by

$$\{\Delta\sigma\} = [\mathbf{D}^e]\{\Delta\epsilon^e\} \quad (6.27)$$

elastoplasticity is described by

$$\{\Delta\sigma\} = [\mathbf{D}^{pl}]\{\Delta\epsilon\} \quad (6.28)$$

where

$$[\mathbf{D}^{pl}] = [\mathbf{D}^e] - [\mathbf{D}^p] \quad (6.29)$$

For perfect plasticity in the absence of hardening or softening it is assumed that once a stress state reaches a failure surface, subsequent changes in stress may shift the stress state to a different position on the failure surface, but not outside it, thus

$$\left\{ \frac{\partial F}{\partial \sigma} \right\}^T \{\Delta\sigma\} = 0 \quad (6.30)$$

Allowing for the possibility of non-associated flow, plastic strain increments occur normal to a plastic potential surface, thus

$$\{\Delta\epsilon^p\} = \lambda \left\{ \frac{\partial Q}{\partial \sigma} \right\} \quad (6.31)$$

Assuming stress changes are generated by elastic strain components only gives

$$\{\Delta\sigma\} = [\mathbf{D}^e] \left( \{\Delta\epsilon\} - \lambda \left\{ \frac{\partial Q}{\partial \sigma} \right\} \right) \quad (6.32)$$

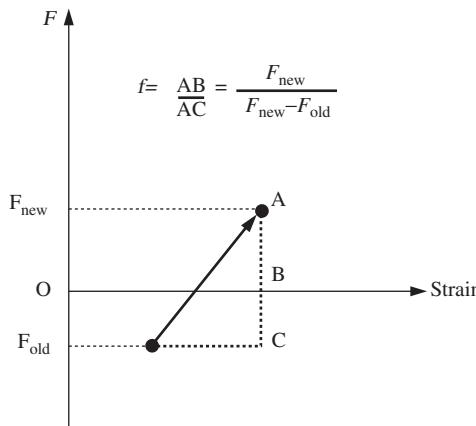
Substitution of equation (6.32) into (6.30) leads to

$$[\mathbf{D}^p] = \frac{[\mathbf{D}^e] \left\{ \frac{\partial Q}{\partial \sigma} \right\} \left\{ \frac{\partial F}{\partial \sigma} \right\}^T [\mathbf{D}^e]}{\left\{ \frac{\partial F}{\partial \sigma} \right\}^T [\mathbf{D}^e] \left\{ \frac{\partial Q}{\partial \sigma} \right\}} \quad (6.33)$$

Explicit versions of  $[\mathbf{D}^p]$  may be obtained for simple failure and potential functions and these are given for von Mises (Yamada *et al.*, 1968) and Mohr–Coulomb (Griffiths and Willson, 1986). See Appendix C for a detailed derivation of (6.33).

The body loads  $\{\mathbf{F}_b\}^i$  in the stress redistribution process are reformed at each iteration by summing the following integral for all elements that possess yielding Gauss points, thus

$$\{\mathbf{F}_b\}^i = \sum_{elements}^{all} \int \int [\mathbf{B}]^T [\mathbf{D}^p] \{\Delta\epsilon\}^i dx dy \quad (6.34)$$



**Figure 6.7** Factoring process for ‘just yielded’ elements

In the event of a loading increment causing a Gauss point to go plastic for the first time, it may be necessary to factor the matrix  $[\mathbf{D}^P]$  in (6.34). A linear interpolation can be used as indicated in Figure 6.7. Thus, instead of using  $[\mathbf{D}^P]$  we use  $f [\mathbf{D}^P]$ , where

$$f = \frac{F_{new}}{F_{new} - F_{old}} \quad (6.35)$$

This simple method represents a forward Euler approach to integrating the elastoplastic rate equations, extrapolating from the point at which the yield surface is crossed. More complicated integrations, which are mainly relevant to tangent stiffness methods, are given in Section 6.9.

Although overshoot of the yield function  $F$  is an integral part of the viscoplastic algorithm, a similar interpolation method to that described by (6.35) can be used if needed to compute the plastic potential derivatives in equation (6.19), using stresses corresponding to  $F \approx 0$ .

## 6.8 Corners on the Failure and Potential Surfaces

For failure and potential surfaces that include ‘corners’ as in Mohr–Coulomb (see Figure 6.6), the derivatives required in equations (6.19) and (6.33) become indeterminate. In the case of the Mohr–Coulomb (or Tresca) surface, this occurs when the angular invariant  $\theta = \pm 30^\circ$ . The method used in the programs to overcome this difficulty is to replace the hexagonal surface by a smooth conical surface if

$$|\sin \theta| > 0.49 \quad (6.36)$$

The conical surfaces are those obtained by substituting either  $\theta = 30^\circ$  or  $\theta = -30^\circ$  into (6.12), depending upon the sign of  $\theta$  as it approaches  $\pm 30^\circ$  (see Appendix C). It should be noted that in the initial stress approach, both the  $F$  and  $Q$  functions must be approximated in this way due to the inclusion of both  $\{\partial Q/\partial \sigma\}$  and  $\{\partial F/\partial \sigma\}$  terms

in (6.33). In the viscoplastic algorithm, however, only the potential function derivatives  $\{\partial Q/\partial\sigma\}$  are approximated since  $\{\partial F/\partial\sigma\}$  is not needed by equation (6.19).

All the programs in this chapter for solving two-dimensional problems have been based on the 8-node quadrilateral element, together with reduced integration (four Gauss points per element). This particular combination has been chosen for its simplicity, and also its well-known ability to compute collapse loads accurately (e.g., Zienkiewicz *et al.*, 1975; Griffiths, 1980, 1982). Of course, other element types could be used if required, by making similar changes to those described in Chapter 5. To alleviate ‘locking’ problems with lower-order elements, ‘reduced integration’ could be used selectively on the volumetric components of the stiffness matrix (see, e.g., Hughes, 1987; Griffiths and Mustoe, 1995).

## Program 6.1 Plane-strain-bearing capacity analysis of an elastic–plastic (von Mises) material using 8-node rectangular quadrilaterals. Flexible smooth footing. Load control. Viscoplastic strain method

```

PROGRAM p61
!-----
! Program 6.1 Plane strain bearing capacity analysis of an elastic-plastic
!           (von Mises) material using 8-node rectangular
!           quadrilaterals. Viscoplastic strain method.
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,incs,iter,iy,k,limit,loaded_nodes,ndim=2,ndof=16,nels,    &
neq,nip=4,nlen,nn,nod=8,nodof=2,nprops=3,np_types,nr,nst=4,nxe,nye
REAL(iwp)::ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,d3=3.0_iwp,d4=4.0_iwp,&
end_time,f,lode_theta,one=1.0_iwp,ptot,sigm,tol,start_time,          &
two=2.0_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,),g(:,),g_g(:,:),g_num(:,:),kdiag(:,),nf(:,:,), &
node(:,),num(:,)
REAL(iwp),ALLOCATABLE::bdylds(:,),bee(:, :,),bload(:, ),coord(:, :,),dee(:, :,), &
der(:, :,),deriv(:, :,),devp(:, ),eld(:, ),eload(:, ),eps(:, ),erate(:, ),evp(:, ), &
evpt(:, :, :,),flow(:, :,),g_coord(:, :,),jac(:, :,),km(:, :,),kv(:, ),loads(:, ), &
m1(:, :,),m2(:, :,),m3(:, :,),oldis(:, ),points(:, :,),prop(:, :,),qinc(:, ), &
sigma(:, ),stress(:, ),tensor(:, :, :,),totd(:, ),val(:, :,),weights(:, ), &
x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen); CALL CPU_TIME(start_time)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),g_g(ndof,nels), &
prop(nprops,np_types),etype(nels),evpt(nst,nip,nels),stress(nst), &
tensor(nst,nip,nels),coord(nod,ndim),jac(ndim,ndim),der(ndim,nod), &
deriv(ndim,nod),g_num(nod,nels),bee(nst,nodof),km(ndof,ndof),eld(ndof), &
eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst), &
devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq))
!-----loop the elements to find global arrays sizes-----

```

```

kdiag=0
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,i7))')                                     &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); kv=zeros
!-----element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    ddt=d4*(one+prop(3,etype(iel)))/(d3*prop(2,etype(iel)))
    IF(ddt<dt) dt=ddt; CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zeros
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1; CALL fspav(kv,km,g,kdiag)
    END DO elements_2
!-----read load weightings and factorise equations-----
READ(10,*) loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes); CALL sparin(kv,kdiag)
!-----load increment loop-----
READ(10,*) tol,limit,incs; ALLOCATE(qinc(incs)); READ(10,*) qinc
WRITE(11,'(/A)')" step      load      disp      iters"
olddis=zeros; totd=zeros; tensor=zeros; ptot=zeros
load_incs: DO iy=1,incs
    ptot=ptot+qinc(iy); iters=0; bdylds=zeros; evpt=zeros
!-----plastic iteration loop-----
    its: DO
        iters=iters+1; loads=zeros
        WRITE(*,'(A,F8.2,A,I4)')" load",ptot," iteration",iters
        DO i=1,loaded_nodes; loads(nf(:,node(i)))=val(i,:)*qinc(iy); END DO
        loads=loads+bdylds; CALL spabac(kv,loads,kdiag)
    !-----check plastic convergence-----
    CALL checon(loads,olddis,tol,converged); IF(iters==1) converged=.FALSE.
    IF(converged.OR.iters==limit) bdylds=zeros
!-----go round the Gauss Points -----
elements_3: DO iel=1,nels
    CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    g=g_g(:,iel); eld=loads(g); bload=zeros
    gauss_pts_2: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
        sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
        CALL invar(stress,sigm,dsbar,lode_theta)
    !-----check whether yield is violated-----
    f=dsbar-SQRT(d3)*prop(1,etype(iel))
    IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
        IF(f>=zero) THEN
            dq1=zeros; dq2=d3/two/dsbar; dq3=zeros
            CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
        END IF
    END DO gauss_pts_2
    END DO elements_3

```

```

        erate=MATMUL(flow,stress); evp=erate*dt
        evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
    END IF
END IF
IF(f>=zero.OR.(converged.OR.iters==limit))THEN
    eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
END IF
!-----update the Gauss Point stresses-----
IF(converged.OR.iters==limit)tensor(:,i,iel)=stress
END DO gauss_pts_2
!-----compute the total bodyloads vector-----
bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
END DO elements_3; IF(converged.OR.iters==limit)EXIT
END DO its; totd=totd+loads
WRITE(11,'(15,2E12.4,15)')iy,ptot,totd(nf(2,node(1))),iters
IF(iters==limit)EXIT
END DO load_incs
CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(totd,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
CALL CPU_TIME(end_time)
WRITE(11,'(A,F12.4)')" Time taken = ",end_time-start_time
STOP
END PROGRAM p61

```

Program 6.1 employs the viscoplastic method to compute the response to loading of an elastic–perfectly plastic von Mises (6.7) material. Plane-strain conditions are enforced and, in order to monitor the load–displacement response, the loads are applied incrementally. As in Progam 4.5, the method uses constant stiffness iterations, thus the relatively time-consuming subroutine `sparin` is called just once, while the subroutine `spabac` is called at each iteration. An outline of the viscoplastic algorithm which comes after the stiffness matrix formation is given in the structure chart in Figure 6.8.

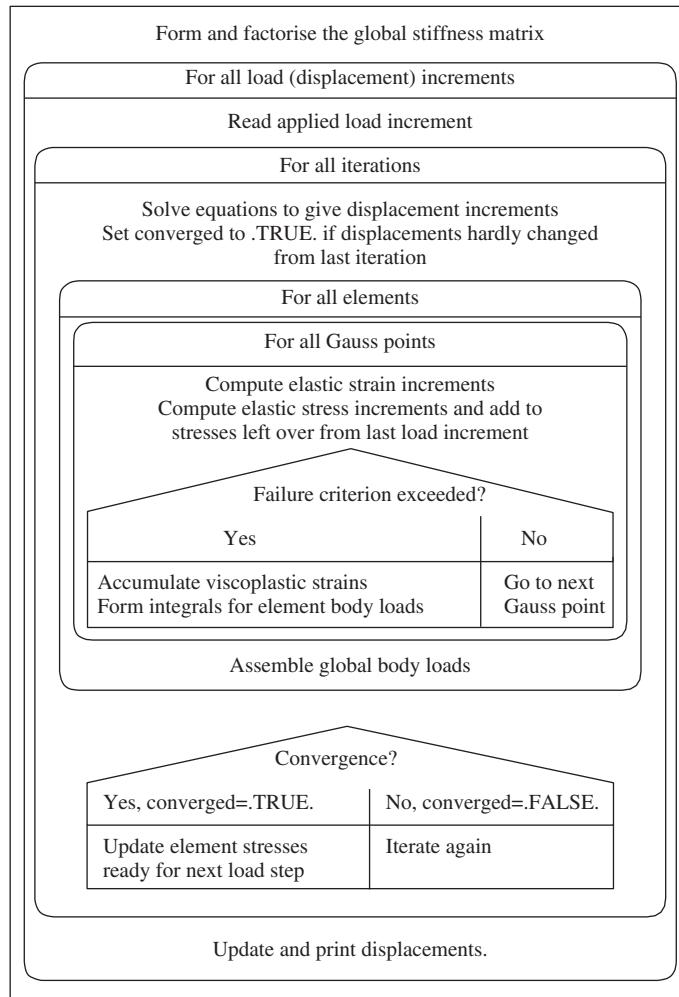
This program uses 8-node quadrilateral elements with numbering in the  $y$ -direction. The familiar subroutine `geom_rect` produces the mesh of rectangular 8-node elements, with the numbering direction, in this case, entered explicitly in the argument list as '`y`'. Subroutines not seen previously include `invar`, which forms the three invariants ( $\sigma_m$ ,  $\bar{\sigma}$ ,  $\theta$ ) [(6.3), (6.4)] from the four Cartesian stress components held in `stress`. It should be noted that in plane-strain plasticity applications, it is necessary to retain four components of stress and strain. Although, by definition,  $\epsilon_z$  must equal zero, the elastic strain in that direction may be non-zero provided

$$\epsilon_z^e = -\epsilon_z^{vp} \quad (6.37)$$

For this reason, the  $4 \times 4$  (2.81) elastic stress–strain matrix  $[\mathbf{D}^e]$  is provided by the subroutine `deemat` with `nst = 4`.

The only other subroutine not encountered before is `formm`, which creates arrays `m1`, `m2` and `m3` used in the calculation of the viscoplastic strain rate from equation (6.25).

The example shown in Figure 6.9 is of half a flexible strip footing (accounting for symmetry) at the surface of a layer of uniform undrained clay. The footing supports a uniform stress,  $q$ , which is increased incrementally to failure. The elastoplastic soil is described by three parameters (`nprops=3`), namely the undrained ‘cohesion’  $c_u$ , followed

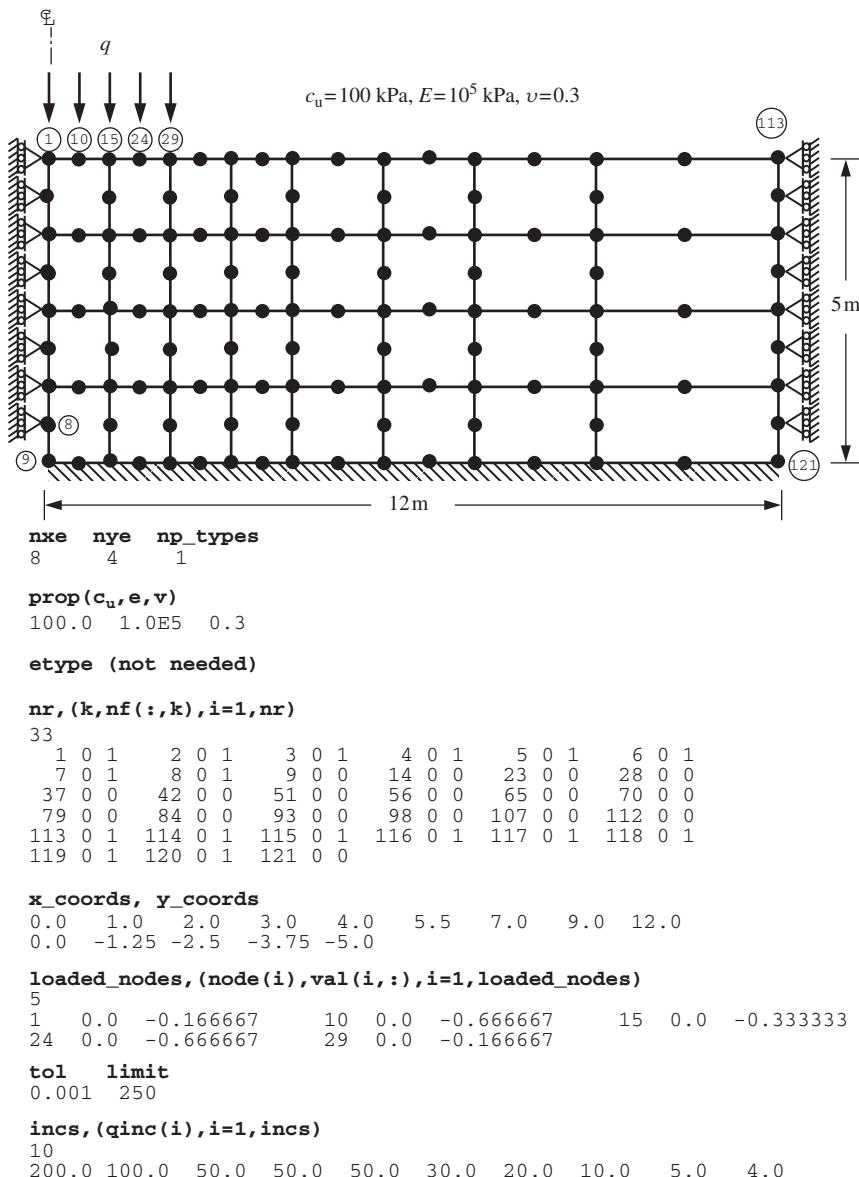


**Figure 6.8** Structure chart for viscoplastic algorithm

by the elastic properties  $E$  and  $v$ . Theoretically, bearing failure in this problem occurs when  $q$  reaches the ‘Prandtl’ load given by

$$q_{ult} = (2 + \pi)c_u \quad (6.38)$$

Apart from the variables `type_2d='plane'`, `element='quadrilateral'`, `nod=8` and `dir='y'`, which are built into the program, the data follows the familiar pattern established in Chapter 5. The ‘loads’ in this case are the nodal forces which would deliver a uniform stress of 1 kN/m<sup>2</sup> across the footing semi-width of 2 m (Appendix A). These ‘weightings’ are then increased proportionally by the load increment values held in the vector `qinc`. In order to capture failure in a load-controlled problem such as this, the increments need to be made smaller as failure is approached. This may



**Figure 6.9** Mesh and data for Program 6.1 example

involve some trial and error on the part of the user in an unfamiliar problem. New input variables involve `tol`, the convergence tolerance (set to 0.001) and `limit`, the iteration ceiling (set to 250) representing the maximum number of iterations (`iters`) that will be allowed within any load increment. If `iters` ever becomes equal to `limit`, the algorithm stops and no more load increments are applied.

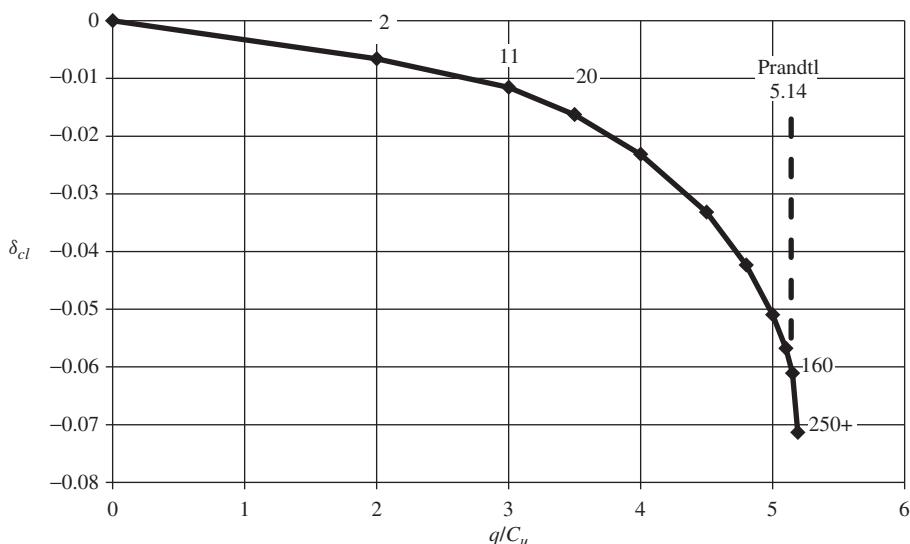
At load levels well below failure, convergence should occur in relatively few iterations. As failure is approached, the algorithm has to work harder and requires more iterations to

```

There are      192 equations and the skyline storage is   4322
step    load        disp     iters
 1  0.2000E+03 -0.6593E-02      2
 2  0.3000E+03 -0.1155E-01     11
 3  0.3500E+03 -0.1630E-01     20
 4  0.4000E+03 -0.2317E-01     33
 5  0.4500E+03 -0.3321E-01     45
 6  0.4800E+03 -0.4235E-01     66
 7  0.5000E+03 -0.5096E-01     82
 8  0.5100E+03 -0.5679E-01    100
 9  0.5150E+03 -0.6109E-01    160
10  0.5190E+03 -0.7135E-01    250
Time taken =          0.4836

```

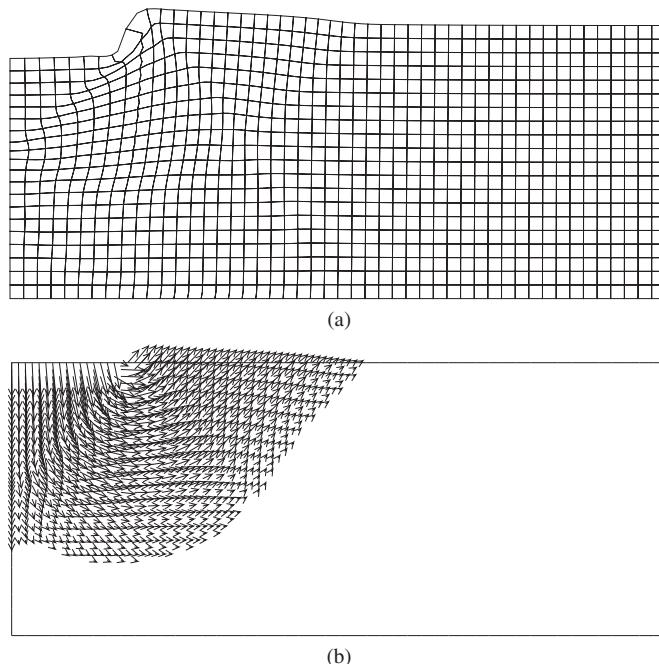
**Figure 6.10** Results from Program 6.1 example



**Figure 6.11** Bearing stress vs. centreline displacement from Program 6.1 example

converge. The computed results for this example are given in Figure 6.10, and show the applied stress, the vertical displacement under the centreline and the number of iterations for convergence. These results have been plotted in Figure 6.11 in the form of a dimensionless bearing capacity factor  $q/c_u$  vs. centreline displacement. The number of iterations to achieve convergence for each load increment is also shown. It is seen that convergence was achieved in 160 iterations when  $q/c_u = 5.15$ , but convergence could not be achieved within the upper limit of 250 when  $q/c_u = 5.19$ . In addition, the displacements are also increasing rapidly at this level of loading, indicating that bearing failure is taking place at a value close to the ‘Prandtl’ load of 5.14. The program also computes and reports the execution time using the intrinsic FORTRAN subroutine CPU\_TIME. It should be noted that all timings quoted in this chapter will vary according to the hardware used.

Program 6.1 creates the graphical PostScript output files \*.msh (undeformed mesh), \*.dis (deformed mesh) and \*.vec (nodal displacement vectors) first encountered in Chapter 5. Although the relatively coarse mesh of Figure 6.9 gave an accurate estimate of



**Figure 6.12** Deformed mesh (a) and nodal vectors (b) at failure from Program 6.1 example

the failure loading, Figure 6.12(a,b) shows the deformed mesh and displacement vectors corresponding to the unconverged ‘solution’ at failure using a rather more refined mesh. The displacements are uniformly magnified to emphasise the deformations. Although the finite element mesh is constrained to remain a continuum, the figures are still able to give a good indication of the form of the failure mechanism.

**Program 6.2 Plane-strain-bearing capacity analysis of an elastic-plastic (von Mises) material using 8-node rectangular quadrilaterals. Flexible smooth footing. Load control. Viscoplastic strain method. No global stiffness matrix assembly. Diagonally preconditioned conjugate gradient solver**

```

PROGRAM p62
! -----
! Program 6.2 Plane strain bearing capacity analysis of an elastic-plastic
!           (von Mises) material using 8-node rectangular
!           quadrilaterals. Viscoplastic strain method. No global
!           stiffness matrix assembly. Diagonally preconditioned
!           conjugate gradient solver.
! -----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,cg_tot,i,iel,incs,iters,iy,k,limit,
loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nlen,nn,nod=8,nodof=2,
nprops=3,np_types,nr,nst=4,nxe,nye
  &
  &
```

```

REAL(iwp)::alpha,beta,cg_tol,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,      &
d3=3.0_iwp,d4=4.0_iwp,end_time,f,lode_theta,one=1.0_iwp,ptot,sigm,      &
start_time,tol,two=2.0_iwp,up,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
LOGICAL::converged,cg_converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_(:, :, :),g_num(:, :, :),nf(:, :, :),node(:, &
num(:)
REAL(iwp),ALLOCATABLE::bdylds(:,bee(:, :, :),bload(:,coord(:, :, :),d(:, &
dee(:, :, :),der(:, :, :),deriv(:, :, :),devp(:,diag_precon(:,eld(:,eload(:, &
eps(:,erate(:,evp(:,evpt(:, :, :, :),flow(:, :, :),g_coord(:, :, :),jac(:, :, :), &
km(:, :, :),loads(:,m1(:, :, :),m2(:, :, :),oldis(:,p(:,points(:, :, :), &
prop(:, :, :),qinc(:,sigma(:,storkm(:, :, :, :),stress(:,tensor(:, :, :, :), &
totd(:,u(:, :, :),weights(:,x(:,xnew(:,x_coords(:,y_coords(:, &
!-----input and initialisation-----
CALL getname(argv,nlen); CALL CPU_TIME(start_time)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),coord(nod,ndim), &
evpt(nst,nip,nels),tensor(nst,nip,nels),etype(nels),jac(ndim,ndim), &
der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),bee(nst,nodof), &
km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof),eload(ndof), &
erate(nst),evp(nst),devp(nst),g(ndof),m1(nst,nst),m2(nst,nst), &
m3(nst,nst),flow(nst,nst),stress(nst),g_g(ndof,nels), &
storkm(ndof,ndof,nels),prop(nprops,np_types))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
WRITE(11,'(A,I7,A)')"There are",neq," equations"
ALLOCATE(loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq),p(0:neq), &
x(0:neq),xnew(0:neq),u(0:neq),diag_precon(0:neq),d(0:neq))
!-----loop to set up global arrays-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
CALL sample(element,points,weights); diag_precon=zero
!-----element stiffness integration, storage and preconditioner-----
elements_2: DO iel=1,nels
    ddt=d4*(one+prop(3,etype(iel)))/(d3*prop(2,etype(iel)))
    if(ddt<dt)dt=ddt; CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1; storkm(:, :, iel)=km
    DO k=1,nodof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
END DO elements_2; diag_precon(1:)=one/diag_precon(1:)
!-----read load weightings-----
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
!-----load increment loop-----
READ(10,*)tol,limit,incs; ALLOCATE(qinc(incs)); READ(10,*)qinc

```

```

      WRITE(11,'(/A)')
      " step    load      disp      iters      cg iters/plastic iter"
      oldis=zero; totd=zero; tensor=zero; ptot=zero; diag_precon(0)=zero
      load_incs: DO iy=1,incs
      ptot=ptot+qinc(iy); iters=0; bdylds=zero; evpt=zero; cg_tot=0
!-----plastic iteration loop-----
      its: DO
      iters=iters+1; loads=zero
      WRITE(*,'(A,F8.2,A,I4)')" load",ptot," iteration",iters
      DO i=1,loaded_nodes; loads(nf(:,node(i)))=val(i,:)*qinc(iy); END DO
      loads=loads+bdylds; d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
      pcg: DO
      cg_iters=cg_iters+1; u=zero
      elements_3 : DO iel=1,nels
      g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
      END DO elements_3
      up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
      loads=loads-u*alpha; d=diag_precon*loads
      beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
      call checon(xnew,x,cg_tol,cg_converged)
      IF(cg_converged.OR.cg_iters==cg_limit)EXIT
      END DO pcg; cg_tot=cg_tot+cg_iters; loads=xnew; loads(0)=zero
!-----check plastic convergence-----
      CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
      IF(converged.OR.iters==limit)bdylds=zero
!-----go round the Gauss Points-----
      elements_4: DO iel=1,nels
      CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
      num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
      g=g_g(:,iel); eld=loads(g); bload=zero
      gauss_pts_2: DO i=1,nip
      CALL shape_der(der,points,i)
      jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
      deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
      eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
      sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
      CALL invar(stress,sigm,dsbar,lode_theta)
!-----check whether yield is violated-----
      f=dsbar-SQRT(d3)*prop(1,etype(iel))
      IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
      IF(f>=zero)THEN
      dq1=zero; dq2=d3/two/dsbar; dq3=zero
      CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
      erate=MATMUL(flow,stress); evp=erate*dt
      evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
      END IF
      END IF
      IF(f>=zero)THEN
      eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
      END IF
!-----update the Gauss Point stresses-----
      IF(converged.OR.iters==limit)tensor(:,i,iel)=stress
      END DO gauss_pts_2
!-----compute the total bodyloads vector-----
      bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
      END DO elements_4; IF(converged.OR.iters==limit)EXIT
      END DO its; totd=totd+loads

```

```

      WRITE(11, '(I5,2E12.4,I5,F17.2)')                                &
      iy,ptot,totd(nf(2,node(1))),iters,REAL(cg_tot)/REAL(iters)
      IF(iters==limit) EXIT
END DO load_incs
CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
CALL CPU_TIME(end_time)
WRITE(11, '(A,F12.4)')" Time taken = ",end_time-start_time
STOP
END PROGRAM p62

```

For non-linear problems, computation times are now more demanding. For example, Program 6.1 required several hundred elastic solutions to reach the failure load. Clearly, vectorised and parallelised algorithms will become essential for much larger problems. The ‘element-by-element’ approach first demonstrated in Program 5.6 will be attractive for large non-linear problems and is implemented in Program 6.2. The data in Figure 6.13 are identical to those used in Program 6.1, with the addition of a conjugate gradient convergence tolerance `cg_tol` set to 0.0001 and conjugate gradient iteration ceiling `cg_limit` set to 100. The results are shown in Figure 6.14, and are nearly identical to those in Figure 6.10. The results table in Figure 6.14 also indicates that approximately 50 conjugate gradient iterations were needed on average for each plastic iteration. In scalar computations, therefore, this algorithm will not be competitive, but it does have some strong attractions for parallel computation. Because a ‘constant stiffness’ approach is being used, groups of elements in Figure 6.9 have constant properties throughout the calculation (there are only four distinct element types in this case). This feature can also be exploited in the parallel algorithms described in Chapter 12. Also, computation costs can be approximately halved by using the previously computed `bldylds` rather than setting `bldylds` to zero at the beginning of each load increment.

```

nxe nye cg_tol cg_limit np_types
8     4    0.0001   100       1

prop(c_u,e,v)
100.0 1.0E5  0.3

etype(not needed)

x_coords, y_coords
0.0   1.0   2.0   3.0   4.0   5.5   7.0   9.0  12.0
0.0  -1.25 -2.5  -3.75 -5.0

loaded_nodes, (node(i),val(i,:),i=1,loaded_nodes)
5
1  0.0  -0.166667    10  0.0  -0.666667    15  0.0  -0.333333
24 0.0  -0.666667   29  0.0  -0.166667

tol limit
0.001 250

incs,(qinc(i),i=1,incs)
10
200.0 100.0  50.0  50.0  50.0  30.0  20.0  10.0   5.0   4.0

```

**Figure 6.13** Data for Program 6.2 example

There are 184 equations

step	load	disp	iters	cg	iters/plastic iter
1	0.2000E+03	-0.6593E-02	2		46.00
2	0.3000E+03	-0.1155E-01	11		46.82
3	0.3500E+03	-0.1630E-01	20		50.45
4	0.4000E+03	-0.2315E-01	33		50.97
5	0.4500E+03	-0.3317E-01	45		52.47
6	0.4800E+03	-0.4228E-01	65		53.34
7	0.5000E+03	-0.5086E-01	81		54.00
8	0.5100E+03	-0.5668E-01	100		53.36
9	0.5150E+03	-0.6090E-01	147		52.65
10	0.5190E+03	-0.7124E-01	250		53.85
Time taken = 4.8984					

**Figure 6.14** Results from Program 6.2 example

### 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

```

PROGRAM p63
!-----
! 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.
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,incs,iters,iy,i3,i4,i5,j,k,limit,nbo2,      &
ndim=2,ndof=16,nels,neq,nip=4,nlen,mn,nod=8,nodof=2,nprops=6,np_types, &
nst=4,nxe,nye
REAL(iwp)::c,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,d3=3.0_iwp,      &
d4=4.0_iwp,d6=6.0_iwp,d180=180.0_iwp,e,f,lode_theta,one=1.0_iwp,pav,    &
penalty=1.0e20_iwp,phi,pi,pr,presc,psi,qq,qs,sigm,snph,                  &
start_dt=1.e15_iwp,tol,two=2.0_iwp,v,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:, ),g(:, ),g_g(:, :, ),g_num(:, :, ),kdiag(:, ),nf(:, :, ), &
no(:, ),node(:, ),num(:, )
REAL(iwp),ALLOCATABLE::bdylds(:, ),bee(:, :, ),bload(:, ),coord(:, :, ),dee(:, :, ), &
der(:, :, ),deriv(:, :, ),devp(:, ),eld(:, ),eload(:, ),eps(:, ),erate(:, ),evp(:, ), &
evpt(:, :, :, ),flow(:, :, ),fun(:, ),gravlo(:, ),g_coord(:, :, ),jac(:, :, ),km(:, :, ), &
kv(:, ),kvc(:, ),loads(:, ),m1(:, :, ),m2(:, :, ),m3(:, :, ),oldis(:, ),points(:, :, ), &
prop(:, :, ),react(:, ),rload(:, ),sigma(:, ),storkv(:, ),stress(:, ),tensor(:, :, :, ), &
totd(:, ),weights(:, ),x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,nbo2,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),      &
x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),g_g(ndof,nels), &

```

```

prop(nprops,np_types),etype(nels),evpt(nst,nip,nels),stress(nst) , &
tensor(nst,nip,nels),coord(nod,ndim),jac(ndim,ndim),der(ndim,nod) , &
deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof) , &
eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst) , &
devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst) , &
fun(nod),rload(ndof))
READ(10,*)prop; etype=1
IF(np_types>1)READ(10,*)((etype(j+(i-1)*nye),i=1,nxe),j=1,nye)
CALL bc_rect(nxe,nye,nf,'y'); neq=MAXVAL(nf)
READ(10,*)qs,x_coords,y_coords
ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq) , &
gravlo(0:neq),react(0:neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); CALL fkdiag(kdiag,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),kvc(kdiag(neq)))
WRITE(11,'(2(A,I7))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
pi=ACOS(-one); dt=start_dt
DO i=1,np_types
    phi=prop(1,i); snph=SIN(phi*pi/d180); e=prop(5,i); v=prop(6,i)
    ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
END DO
CALL sample(element,points,weights); gravlo=zero; kv=zero
!-----element stiffness integration and gravity loads---
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    km=zeros; eld=zeros
    gauss_pts_2: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        eld(2:ndof:2)=eld(2:ndof:2)+fun(:)*det*weights(i)
    END DO gauss_pts_2
    CALL fsparv(kv,km,g,kdiag); gravlo(g)=gravlo(g)-eld*prop(4,etype(iel))
END DO elements_2; kvc=kv
!-----surcharge loads-----
DO i=1,nxe
    i3=g_num(3,(i-1)*nye+1); i4=g_num(4,(i-1)*nye+1)
    i5=g_num(5,(i-1)*nye+1); qq=(x_coords(i+1)-x_coords(i))*qs
    gravlo(nf(2,i3))=gravlo(nf(2,i3))-qq/d6
    gravlo(nf(2,i4))=gravlo(nf(2,i4))-qq*two/d3
    gravlo(nf(2,i5))=gravlo(nf(2,i5))-qq/d6
END DO
!-----factorise equations-----
CALL sparin(kv,kdiag); CALL spabac(kv,gravlo,kdiag); gravlo(0)=zero
!-----set up initial stresses-----
elements_3: DO iel=1,nels
    CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel))); g=g_g(:,iel)
    eld=gravlo(g); num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    int_pts_2: DO i=1,nip

```

```

CALL shape_der(der,points,i); jac=MATMUL(der,coord); CALL invert(jac)
deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
sigma=MATMUL(dee,MATMUL(bee,eld))
tensor(1,i,iel)=sigma(1); tensor(2,i,iel)=sigma(2)
tensor(3,i,iel)=sigma(3); tensor(4,i,iel)=sigma(4)
END DO int_pts_2
END DO elements_3
!-----fixed displacement data and factorise equations-----
fixed_freedoms=2*nbo2+1
ALLOCATE(node(fixed_freedoms),no(fixed_freedoms),storkv(fixed_freedoms))
node(1)=1; k1
DO i=1,nbo2; k=k+(2*nye)+1; node(2*i)=k; k=k+nye+1; node(2*i+1)=k; END DO
DO i=1,fixed_freedoms; no(i)=nf(2,node(i)); END DO
kv=kvc; kv(kdiag(no))=kv(kdiag(no))+penalty; storkv=kv(kdiag(no))
CALL sparin(kv,kdiag)
!-----load increment loop-----
READ(10,*)tol,limit,incs,presc; oldis=zero; totd=zero
WRITE(11,'(/A)')" step disp load1 load2 iters"
disp_incs: DO iy=1,incs
  iters=0; bdylds=zero; react=zero; evpt=zero
!-----plastic iteration loop-----
  its: DO; iters=iters+1
    WRITE(*,'(A,E11.3,A,I4)')" disp",iy*presc," iteration",iters
    loads=zero; loads=loads+bdylds
    DO i=1,fixed_freedoms; loads(no(i))=storkv(i)*presc; END DO
    CALL spabac(kv,loads,kdiag)
!-----check plastic convergence-----
    CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
    IF(converged.OR.iters==limit)bdylds=zero
!-----go round the Gauss Points -----
elements_4: DO iel=1,nels
  phi=prop(1,etype(iel)); c=prop(2,etype(iel)); psi=prop(3,etype(iel))
  CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel)))
  bload=zero; rload=zero; num=g_num(:,iel)
  coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
  gauss_pts_4: DO i=1,nip
    CALL shape_der(der,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
    sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
    CALL invar(stress,sigm,dsbar,lode_theta)
!-----check whether yield is violated-----
    CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
    IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
      IF(f>=zero)THEN
        CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
        CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
        erate=MATMUL(flow,stress); evp=erate*dt
        evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
      END IF
    END IF
    IF(f>=zero.OR.(converged.OR.iters==limit))THEN
      eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
    END IF
!-----update the Gauss Point stresses-----
  END DO
END DO

```

```

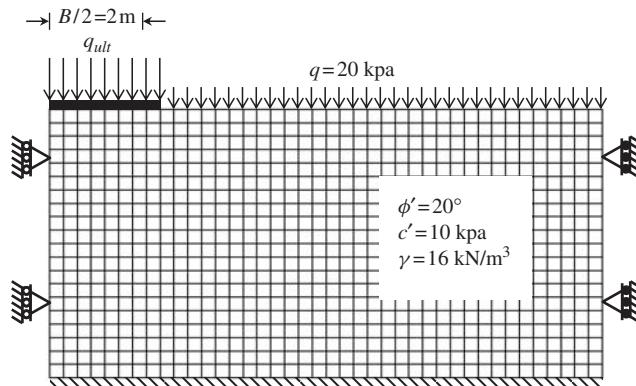
        IF(converged.OR.iters==limit)THEN; tensor(:,i,iel)=stress
          rload=rload+MATMUL(stress,bee)*det*weights(i)
        END IF
      END DO gauss_pts_4
!-----compute the total bodyloads vector-----
      bdylds(g)=bdylds(g)+bload; react(g)=react(g)+rload; bdylds(0)=zero
    END DO elements_4
    IF(converged.OR.iters==limit)EXIT
  END DO its
  totd=totd+loads; pr=zero
  DO i=1,fixed_freedoms; pr=pr+react(no(i)); END DO
  pr=pr/x_coords(nbo2+1); pav=zero
  DO i=1,nbo2
    pav=pav+tensor(2,1,(i-1)*nye+1)+tensor(2,2,(i-1)*nye+1)
  END DO; pav=pav/(two*nbo2)
  WRITE(11,'(I5,3E12.4,I5)')iy,-totd(1),-pr,-pav,iters
  IF(iters==limit)EXIT
END DO disp_incs
CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(totd,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p63

```

The first example in this chapter was of a smooth flexible strip footing on an undrained clay using a von Mises failure criterion. In view of the importance of bearing capacity and settlement in geotechnical engineering, we now include a second bearing capacity analysis program involving a strip footing on a Mohr–Coulomb material. This time the footing is smooth and rigid, so displacement control is employed similar to that used in the elastic example of Figure 5.11. Since the soil strength is governed by friction and cohesion, this program includes the option of soil self-weight and a surface surcharge. The example and data shown in Figure 6.15 is of a rectangular mesh of 8-node square elements, generated as usual using the subroutine `geom_rect`.

The numbers of elements in the  $x$ - and  $y$ -directions (`nxe` and `nye`) are read, followed by `nbo2` which is the number of elements to be rigidly displaced in the bearing capacity analysis. Due to symmetry, only half the footing ( $B/2$ ) is modelled. The data then reads six material properties into `prop` as  $(c', \phi', \psi, \gamma, E$  and  $\nu)$  followed by the surface surcharge `qs`. Multiple material property groups can be included, however, `etype` data is not needed in this example since `np_types=1`. Although the program numbers the nodes and elements in the ' $y$ ' direction for storage efficiency, the `etype` data, if needed, should be read in row by row starting in the top left corner. A further customisation of this program for footing analysis is that the boundary conditions data, normally read into the `nf` array, is generated automatically by the subroutine `bc_rect` (see Appendix E) which delivers roller boundary conditions on the left and right sides and full fixity at the base of the mesh. Input continues with the `x_coords` and `y_coords` data, the tolerance `tol`, the limit for plastic iterations and finally, the number `incs` and magnitude `presc` of the incremental vertical displacements to be applied to the rigid footing.

Subroutine `mocouf` computes the Mohr–Coulomb failure function  $F$  from the current stress state and subroutine `mocouq` forms the derivatives of the Mohr–Coulomb potential function  $Q$  with respect to the three stress invariants and these values are held in `dq1`, `dq2` and `dq3`. In Programs 6.1 and 6.2, similar subroutines corresponding to the von Mises



```

nx e   ny e   nbo2    np_types
40     20      8        1

prop (phi,c,psi,gamma,e,v)
20.0 10.0 20.0   16.0 1.0E5  0.3

etype (not needed)

qs
20.0

x_coords, y_coords
0.0 0.25 0.5 0.75 1.0 1.25 1.5 1.75 2.0 2.25
2.5 2.75 3.0 3.25 3.5 3.75 4.0 4.25 4.5 4.75
5.0 5.25 5.5 5.75 6.0 6.25 6.5 6.75 7.0 7.25
7.5 7.75 8.0 8.25 8.5 8.75 9.0 9.25 9.5 9.75 10.0

5.0 4.75 4.5 4.25 4.0 3.75 3.5 3.25 3.0 2.75
2.5 2.25 2.0 1.75 1.5 1.25 1.0 0.75 0.5 0.25  0.0

tol    limit   incs   presc
0.001   250     25    -0.001

```

**Figure 6.15** Mesh and data for Program 6.3 example

criterion could have been created, but the required expressions were so trivial that they were written directly into the main program.

For each material type, six properties must be read in (`nprops=6`); the friction angle  $\phi$ , the cohesion  $c$ , the dilation angle  $\psi$ , the total unit weight  $\gamma$ , Young's modulus  $E$  and Poisson's ratio  $\nu$ .

The mesh consists of 40 elements in the  $x$ -direction and 20 in the  $y$ -direction, with a half-footing width covering eight elements. The soil properties are  $\phi' = 20^\circ$ ,  $c' = 10 \text{ kPa}$ ,  $\psi = 20^\circ$ ,  $\gamma = 16 \text{ kN/m}^3$ ,  $E = 1 \times 10^5 \text{ kPa}$  and  $\nu = 0.3$ , with a ground surface surcharge of  $q = 20 \text{ kPa}$ . The mesh consists of square elements of side length 0.25 m, so the full footing width is given by  $B = 4 \text{ m}$ . The plasticity convergence tolerance is read as 0.001 and the iteration ceiling set to 250. Finally, the data calls for 25 displacement increments of  $-0.001$ .

There are 4800 equations and the skyline storage is 495810

step	disp	load1	load2	iters
1	0.1000E-02	0.5733E+02	0.5497E+02	11
2	0.2000E-02	0.8502E+02	0.8249E+02	22
3	0.3000E-02	0.1112E+03	0.1084E+03	25
4	0.4000E-02	0.1366E+03	0.1336E+03	27
5	0.5000E-02	0.1613E+03	0.1581E+03	30
6	0.6000E-02	0.1853E+03	0.1818E+03	33
7	0.7000E-02	0.2086E+03	0.2048E+03	36
8	0.8000E-02	0.2313E+03	0.2272E+03	40
9	0.9000E-02	0.2533E+03	0.2490E+03	57
10	0.1000E-01	0.2744E+03	0.2700E+03	81
11	0.1100E-01	0.2944E+03	0.2900E+03	120
12	0.1200E-01	0.3129E+03	0.3084E+03	139
13	0.1300E-01	0.3296E+03	0.3249E+03	149
14	0.1400E-01	0.3440E+03	0.3392E+03	157
15	0.1500E-01	0.3562E+03	0.3512E+03	162
16	0.1600E-01	0.3663E+03	0.3611E+03	166
17	0.1700E-01	0.3742E+03	0.3689E+03	172
18	0.1800E-01	0.3788E+03	0.3734E+03	184
19	0.1900E-01	0.3795E+03	0.3739E+03	197
20	0.2000E-01	0.3796E+03	0.3740E+03	199
21	0.2100E-01	0.3797E+03	0.3741E+03	199
22	0.2200E-01	0.3798E+03	0.3741E+03	200
23	0.2300E-01	0.3798E+03	0.3741E+03	200
24	0.2400E-01	0.3799E+03	0.3741E+03	200
25	0.2500E-01	0.3799E+03	0.3741E+03	200

**Figure 6.16** Results from Program 6.3 example

The results shown in Figure 6.16 give the displacement of the footing followed by the average pressure on the footing calculated by two different methods. The column marked *load1* gives the sum of the reaction forces computed using the converged stress field as

$$\{\mathbf{P}\}_r = \sum_{elements}^{all} \int \int [\mathbf{B}]^T \{\boldsymbol{\sigma}\} dx dy \quad (6.39)$$

divided by the footing width, and the column marked *load2* gives the average vertical stress  $\sigma_y$  in the row of Gauss points just beneath the displaced nodes. The final column marked *iters* gives the number of iterations needed for convergence. The stress averaging *load2* gives a slightly lower value due to the avoidance of shear concentrations at the footing edge and is generally preferred. It can be seen that as the footing is displaced, the pressure under the displaced nodes gradually increases and eventually flattens out, as shown plotted for the stress averaging option in Figure 6.17.

The ultimate bearing capacity is estimated to be  $q_u = 374$  kPa, which is in excellent agreement with alternative solutions such as that of Martin (2004), who predicts  $q_u = 368$  kPa using methods of characteristics. The nodal displacement vectors at failure are shown in Figure 6.18.

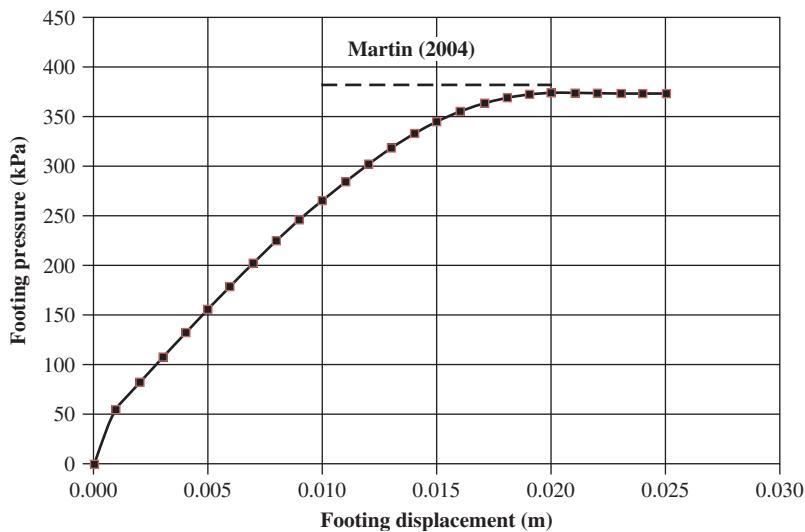


Figure 6.17 Footing pressure vs. footing displacement from Program 6.3 example

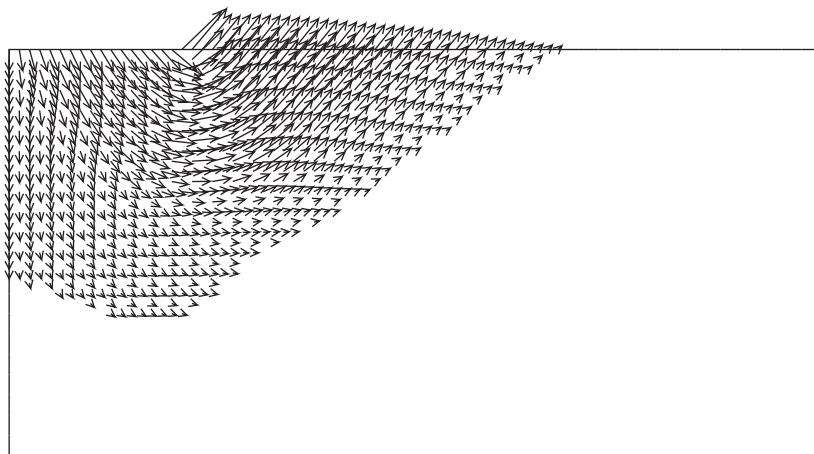


Figure 6.18 Displacement vectors at failure from Program 6.3 example

#### 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

```

PROGRAM p64
! -----
! Program 6.4 Plane strain slope stability analysis of an elastic-plastic
! (Mohr-Coulomb) material using 8-node rectangular
! quadrilaterals. Viscoplastic strain method.
!
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)

```

```

INTEGER::i,iel,iters,iy,limit,ndim=2,ndof=16,nels,neq,nip=4,nlen,nn,      &
nod=8,nodof=2,nprops=6,np_types,nsrf,nst=4,nx1,nx2,nye,ny1,ny2
REAL(iwp)::cf,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,d4=4.0_iwp,      &
d180=180.0_iwp,e,f,fmax,h1,h2,lode_theta,one=1.0_iwp,phi,phif,pi,psi,    &
psif,sigm,snph,start_dt=1.e15_iwp,s1,tnph,tnps,tol,two=2.0_iwp,v,w1,w2,&
zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:)),g_num(:,:),kdiag(:,nf(:,,:), &
num(:)
REAL(iwp),ALLOCATABLE::bdylds(:,bee(:,bload(:,coord(:,dee(:,&
devp(:,elastic(:,eld(:,eload(:,eps(:,erate(:,evp(:,evpt(:,&
flow(:,fun(:,gravlo(:,g_coord(:,km(:,kv(:,loads(:,m1(:,&
m2(:,m3(:,oldis(:,points(:,prop(:,sigma(:,srf(:,        &
weights(:,&
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)w1,s1,w2,h1,h2,nx1,nx2,ny1,ny2,np_types; nye=ny1+ny2
nels=nx1*nye+ny2*nx2; nn=(3*nye+2)*nx1+2*nye+1+(3*ny2+2)*nx2
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),      &
num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod),      &
g_g(ndof,nels),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof),      &
eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst),      &
devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),      &
prop(nprops,np_types),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
CALL emb_2d_bc(nx1,nx2,ny1,ny2,nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),      &
gravlo(0:neq),elastic(0:neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL emb_2d_geom(iel,nx1,nx2,ny1,ny2,w1,s1,w2,h1,h2,coord,num)
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,i7))')                                     &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); kv=zero; gravlo=zero
!-----element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; eld=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i)
        CALL bee8(bee,coord,points(i,1),points(i,2),det)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        eld(nodof:nodof:nodof)=eld(nodof:nodof:nodof)+fun(:)*det*weights(i)
    END DO gauss_pts_1
    CALL fsparv(kv,km,g,kdiag); gravlo(g)=gravlo(g)-eld*prop(4,etype(iel))
END DO elements_2
!-----factorise equations-----
CALL sparin(kv,kdiag); pi=ACOS(-one)
!-----trial strength reduction factor loop-----
READ(10,*)tol,limit,nsrf; ALLOCATE(srf(nsrf)); READ(10,*)srf
WRITE(11,'(/A)')" srf      max disp  iters"
srf_trials: DO iy=1,nsrf
    dt=start_dt

```

```

DO i=1,np_types
    phi=prop(1,i); tnph=TAN(phi*pi/d180); phif=ATAN(tnph/srf(iy))
    snph=SIN(phif); e=prop(5,i); v=prop(6,i)
    ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt) dt=ddt
END DO; iters=0; bdylds=zero; evpt=zero; oldis=zero
!-----plastic iteration loop-----
its: DO
    fmax=zero; iters=iters+1; loads=gravlo+bdylds
    CALL spabac(kv,loads,kdiag); loads(0)=zero
    IF(iy==1.AND.iters==1)elastic=loads
!-----check plastic convergence-----
    CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
    IF(converged.OR.iters==limit)bdylds=zero
!-----go round the Gauss Points -----
elements_3: DO iel=1,nels
    bload=zero; phi=prop(1,etype(iel)); tnph=TAN(phi*pi/d180)
    phif=ATAN(tnph/srf(iy))*d180/pi; psi=prop(3,etype(iel))
    tnps=TAN(psi*pi/d180); psif=ATAN(tnps/srf(iy))*d180/pi
    cf=prop(2,etype(iel))/srf(iy); e=prop(5,etype(iel))
    v=prop(6,etype(iel)); CALL deemat(dee,e,v); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
    gauss_pts_2: DO i=1,nip
        CALL bee8(bee,coord,points(i,1),points(i,2),det)
        eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel); sigma=MATMUL(dee,eps)
        CALL invar(sigma,sigm,dsbar,lode_theta)
!-----check whether yield is violated-----
        CALL mocouf(phif,cf,sigm,dsbar,lode_theta,f); IF(f>fmax)fmax=f
        IF(converged.OR.iters==limit)THEN; devp=sigma; ELSE
            IF(f>=zero.OR.(converged.OR.iters==limit))THEN
                CALL mocouq(psif,dsbar,lode_theta,dq1,dq2,dq3)
                CALL formm(sigma,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
                erate=MATMUL(flow,sigma); evp=erate*dt
                evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
            END IF
        END IF
        IF(f>=zero)THEN
            eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
        END IF
    END DO gauss_pts_2
!-----compute the total bodyloads vector-----
    bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
END DO elements_3
WRITE(*,'(A,F7.2,A,I4,A,F8.3)') &
    " srf",srf(iy)," iteration",iters," F_max",fmax
IF(converged.OR.iters==limit)EXIT
END DO its; WRITE(11,'(F7.2,E12.4,I5)')srf(iy),MAXVAL(ABS(loads)),iters
IF(iters==limit)EXIT
END DO srf_trials
CALL dismsh(loads-elastic,nf,0.1_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads-elastic,nf,0.1_iwp,0.25_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p64

```

This program is, in many ways, similar to Program 6.3. The problem to be analysed is a slope of Mohr–Coulomb material subjected to gravity loading. The factor of safety (*FS*) of the slope is to be assessed, and this quantity is defined as the proportion by which  $\tan \phi$  and *c* must be reduced in order to cause failure with the gravity loading held constant (Zienkiewicz *et al.*, 1975). This is in contrast to the previous programs in this

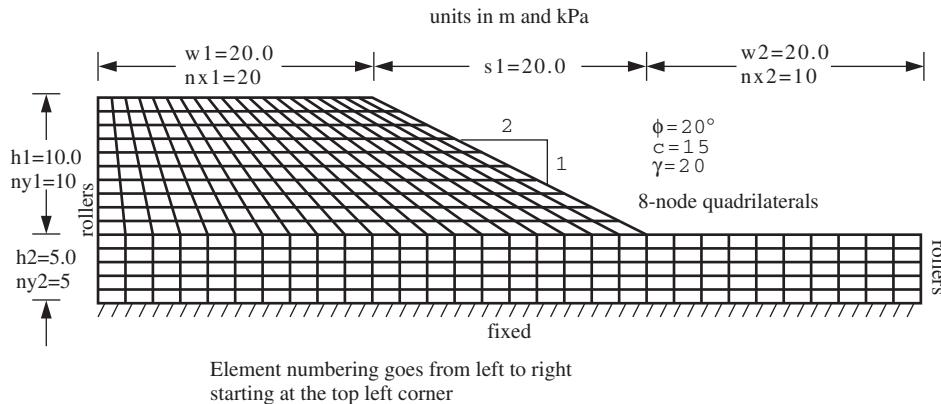
chapter, in which failure was induced by increasing the loads or deformations with the material properties remaining constant.

Gravity loads are generated in the manner described in Chapter 5 (Program 5.4) and applied to the slope in a single increment. A trial strength-reduction factor loop gradually weakens the soil until the algorithm fails to converge. Each entry of this loop implements a gradually increasing strength reduction factor  $SRF$  which is read in as data. The factored soil strength parameters that go into the elastoplastic analysis are obtained from

$$\begin{aligned}\phi_f &= \arctan(\tan \phi / SRF) \\ c_f &= c / SRF\end{aligned}\quad (6.40)$$

Several gradually increasing values of the  $SRF$  factor are attempted until the algorithm fails to converge. The smallest value of  $SRF$  to cause failure is then interpreted as the factor of safety  $FS$ . For a detailed description of the algorithm, the reader is referred to Griffiths and Lane (1999).

Subroutines new to this program include `emb_2d_geom` and `emb_2d_bc`. These subroutines generate the mesh and boundary conditions for a standard slope cross-section of the type shown in Figure 6.19, with dimensions and mesh density controlled through



```

w1      s1      w2      h1      h2
20.0    20.0    20.0   10.0    5.0

nx1    nx2    ny1    ny2
20      10     10     5

np_types
1

prop(phi,c,psi,gamma,e,v)
20.0  15.0  0.0  20.0  1.0e5  0.3

etype(not needed)

tol      limit
0.0001  500

nsrf,(srf(i),i=1,nsrf)
6
1.0  1.2  1.4  1.5  1.55  1.6

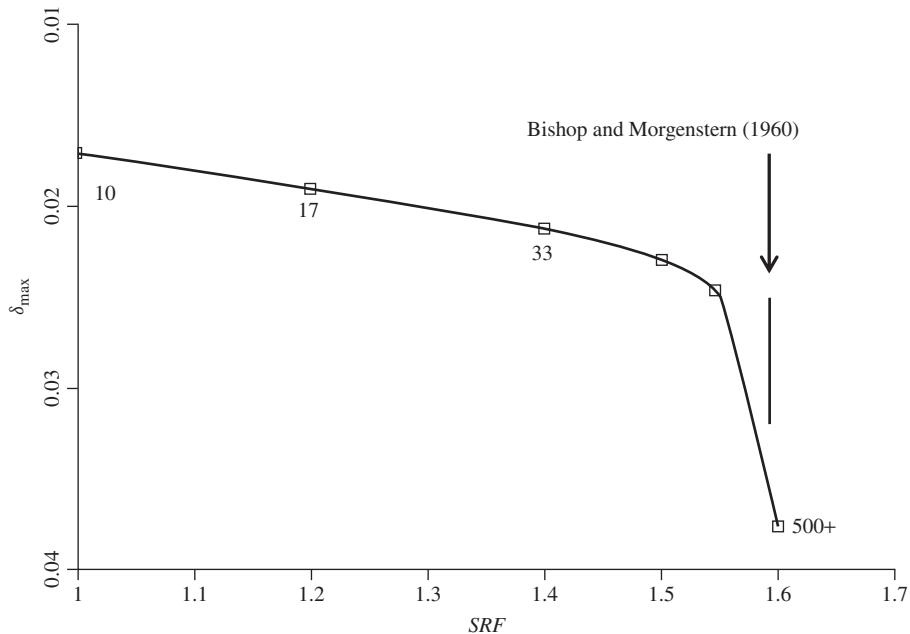
```

**Figure 6.19** Mesh and data for Program 6.4 example

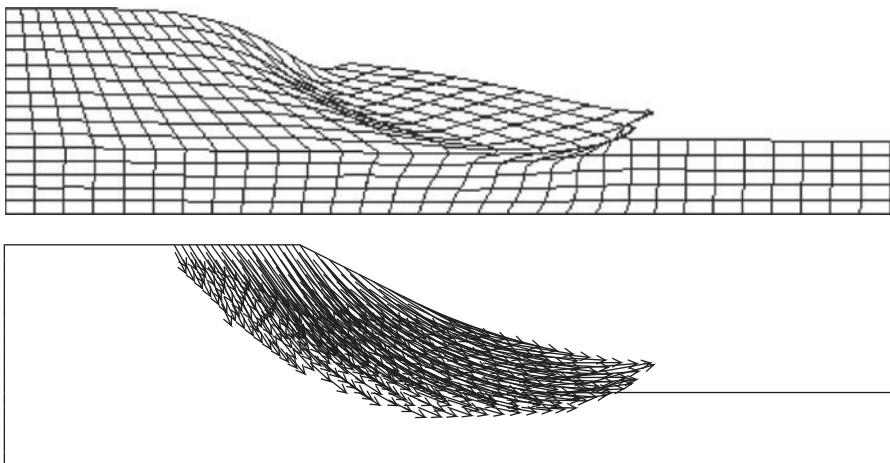
There are 2120 equations and the skyline storage is 151000

srf	max disp	iters
1.00	0.1711E-01	10
1.20	0.1889E-01	17
1.40	0.2115E-01	33
1.50	0.2283E-01	67
1.55	0.2446E-01	244
1.60	0.3761E-01	500

**Figure 6.20** Results from Program 6.4 example



**Figure 6.21** Maximum nodal displacement vs. SRF from Program 6.4 example



**Figure 6.22** Deformed mesh and displacement vectors at failure from Program 6.4 example

the input data. The boundary conditions are rollers to the left and right vertical boundaries, and full fixity at the base.

Figure 6.19 shows the mesh and data for an analysis of a homogeneous 2:1 slope with  $\phi = 20^\circ$  and  $c = 15 \text{ kN/m}^2$ . The dilation angle  $\psi$  is set to zero and the unit weight is given as  $\gamma = 20 \text{ kN/m}^3$ . The elastic parameters are given nominal values of  $E = 1 \times 10^5 \text{ kN/m}^2$  and  $\nu = 0.3$  since they have little influence on the computed factor of safety. The convergence tolerance and iteration ceiling are set to  $\text{tol}=0.0001$  and  $\text{limit}=500$ , respectively. Six trial strength-reduction factors ( $\text{nsrf}=6$ ) are input, ranging from 1.0 to 1.6.

No `etype` data is required in this homogeneous example, but if it is required, the user needs to know that element numbering proceeds row by row, starting at the top left corner of the mesh.

The output in Figure 6.20 gives the strength-reduction factor, the maximum nodal displacement at convergence and the number of iterations to achieve convergence. It can be seen that when `srf=1.6`, the iteration ceiling of 500 was reached. A plot of these results in Figure 6.21 shows that the displacements increase rapidly at this level of strength reduction, indicating a factor of safety of about 1.6. Bishop and Morgenstern's charts (1960) give a factor of safety of 1.593 for the slope under consideration. Figure 6.22 displays the PostScript files `*.dis` and `*.vec`, which show the deformed mesh and displacement vectors corresponding to slope failure. The mechanism of failure is clearly shown to be of the 'toe' type.

## Program 6.5 Plane-strain earth pressure analysis of an elastic–plastic (Mohr–Coulomb) material using 8-node rectangular quadrilaterals. Rigid smooth wall. Initial stress method

```
PROGRAM p65
!-----
! Program 6.5 Plane strain earth pressure analysis of an elastic-plastic
!           (Mohr-Coulomb) material using 8-node rectangular
!           quadrilaterals. Initial stress method.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,incs,iter,iy,k,limit,ndim=2,ndof=16,nels, &
neq,nip=4,nlen,nn,nod=8,nodof=2,nprops=7,np_types,nr,nst=4,nxe,nye
REAL(iwp)::c,det,dsbar,e,f,fac,fnew,gamma,k0,lode_theta,one=1.0_iwp,ot, &
pav,phi,psi,pr,presc,pt5=0.5_iwp,sigm,penalty=1.0e20_iwp,tol,v, &
zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g_g(:, :,g_num(:, :,kdiag(:, nf(:, :, &
no(:, node(:, num(:, sense(:)
REAL(iwp),ALLOCATABLE::bdylds(:,bee(:, :,bload(:,coord(:, :,dee(:, :, &
der(:, :,deriv(:, :,eld(:,eload(:,elso(:,eps(:,fun(:,gc(:, &
g_coord(:, :,jac(:, :,km(:, :,kv(:,loads(:,oldis(:,pl(:, :, &
points(:, :,prop(:, :,react(:,rload(:,sigma(:,storkv(:,stress(:, &
tensor(:, :,totd(:,weights(:,x_coords(:,y_coords(:, &
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
```

```

READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),fun(nod), &
tensor(nst,nip,nels),g_g(ndof,nels),coord(nod,ndim),stress(nst), &
jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),g_num(nod,nels), &
bee(nst,ndof),km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof), &
eload(ndof),pl(nst,nst),elso(nst),g(ndof),gc(ndim),rload(ndof), &
prop(nprops,np_types),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formmf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq), &
react(0:neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq))) &
WRITE(11,'(2(A,I7))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); tensor=zero; kv=zero
!-----element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(6,etype(iel)),prop(7,etype(iel)))
    gamma=prop(4,etype(iel)); k0=prop(5,etype(iel))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); gc=MATMUL(fun,coord)
        tensor(2,i,iel)=(gc(2)-y_coords(1))*gamma
        tensor(1,i,iel)=(gc(2)-y_coords(1))*gamma*k0
        tensor(4,i,iel)=tensor(1,i,iel); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read displacement data and factorise equations---
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),no(fixed_freedoms),&
    storkv(fixed_freedoms))
    READ(10,*)(node(i),sense(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; storkv=kv(kdiag(no))
END IF; CALL sparin(kv,kdiag)
!-----displacement increment loop-----
READ(10,*)tol,limit,incs,presc
WRITE(11,'(/A)')
    " step      disp      load(av)      load(react)      moment      iters"
    oldis=zero; totd=zero; bdylds=zero
    disp_incs: DO iy=1,incs
        iters=0; react=zero
    !-----plastic iteration loop-----
    its: DO
        iters=iters+1; loads=bdylds
        WRITE(*,'(A,E11.3,A,I4)')" disp",iy*presc, " iteration",iters

```

```

DO i=1,fixed_freedoms; loads(nf(1,node(i)))=storkv(i)*presc; END DO
CALL spabac(kv,loads,kdiag); bdylds=zero
!-----check plastic convergence-----
CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
!-----go round the Gauss Points -----
elements_3: DO iel=1,nels
    phi=prop(1,etype(iel)); c=prop(2,etype(iel)); psi=prop(3,etype(iel))
    e=prop(6,etype(iel)); v=prop(7,etype(iel)); CALL deemat(dee,e,v)
    bload=zero; rload=zero; num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
    gauss_pts_2: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
        sigma=MATMUL(dee,eps); stress=sigma+tensor(:,i,iel)
        CALL invar(stress,sigm,dsbar,lode_theta)
    !-----check whether yield is violated-----
    CALL mocouf(phi,c,sigm,dsbar,lode_theta,fnew); else=zero
    IF(fnew>zero)THEN
        stress=tensor(:,i,iel); CALL invar(stress,sigm,dsbar,lode_theta)
        CALL mocouf(phi,c,sigm,dsbar,lode_theta,f); fac=fnew/(fnew-f)
        stress=(one-fac)*sigma+tensor(:,i,iel)
        CALL mcdpl(phi,psi,dee,stress,pl); pl=fac*pl
        else=MATMUL(pl,eps); eload=MATMUL(else,bee)
        bload=bload+eload*det*weights(i)
    END IF
!-----update the Gauss Point stresses-----
    IF(converged.OR.iters==limit)THEN
        tensor(:,i,iel)=tensor(:,i,iel)+sigma-else
        rload=rload+MATMUL(tensor(:,i,iel),bee)*det*weights(i)
    END IF
    END DO gauss_pts_2
!-----compute the total bodyloads vector -----
    bdylds(g)=bdylds(g)+bload; react(g)=react(g)+rload
    bdylds(0)=zero; react(0)=zero
    END DO elements_3; IF(converged.OR.iters==limit)EXIT
END DO its; totd=totd+loads; pr=zero; ot=zero; pav=zero
DO i=1,fixed_freedoms
    pr=pr+react(no(i)); ot=ot+react(no(i))*g_coord(2,node(i))
END DO
DO i=1,4
    pav=pav+(y_coords(i)-y_coords(i+1))*(tensor(1,1,i)+tensor(1,3,i))*pt5
END DO
WRITE(11,'(I5,4E12.4,I5)')iy,iy*presc,-pav,pr,ot,iters
IF(iters==limit)EXIT
END DO disp_incs
CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(totd,nf,0.05_iwp,0.5_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p65

```

The initial stress method of stress redistribution is demonstrated in a problem of passive earth pressure, in which a smooth wall is translated into a bed of cohesionless soil. As in Program 6.1, a rectangular mesh of 8-noded elements is generated with nodes and freedoms counted in the y-direction. An additional feature of this program which appears in the element integration and assembly section is the generation of starting self-weight ‘at

rest' stresses. The coordinates of each Gauss point are calculated using the isoparametric property

$$\begin{aligned} x &= \sum_{i=1}^8 N_i x_i \\ y &= \sum_{i=1}^8 N_i y_i \end{aligned} \quad (6.41)$$

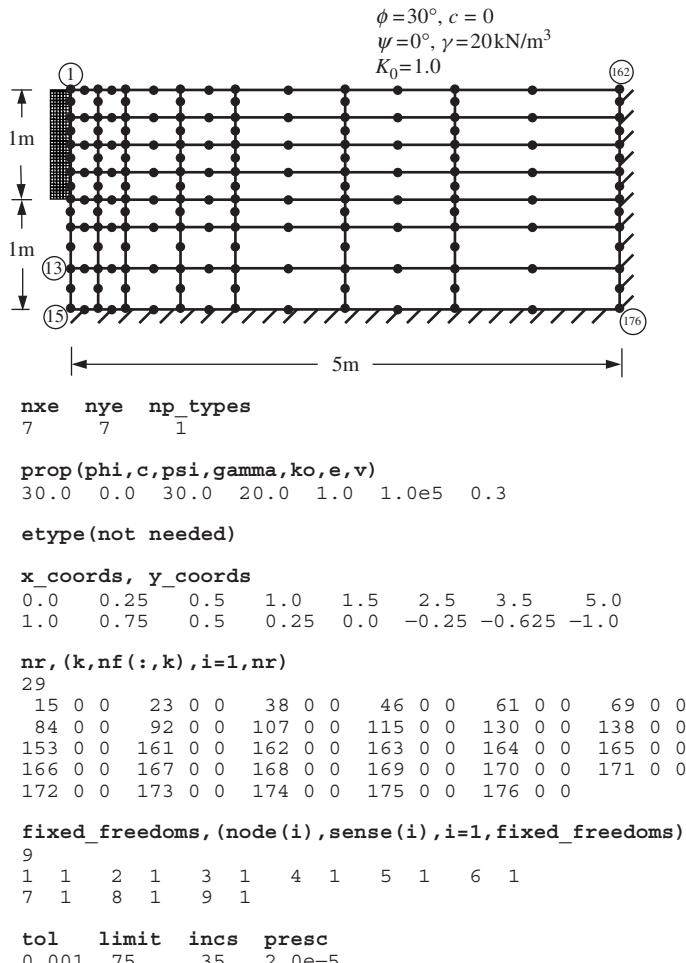
with the  $x$ - and  $y$ -coordinates that result held in the one-dimensional array `gc(1)` and `gc(2)`, respectively. Only the  $y$ -coordinate is required in this case, and the vertical stress  $\sigma_y$  is obtained after multiplication by the unit weight held in `gamma`. The horizontal effective stresses  $\sigma_x$  and  $\sigma_z$  are obtained by multiplying  $\sigma_y$  by the 'at rest' earth pressure coefficient  $K_0$  held in `k0`.

Data relating to the geometry and boundary conditions follow a familiar course. After the stiffness matrix formulation, the nodes and senses of the freedoms which are to receive prescribed displacements are read, followed by the plastic convergence tolerance `tol`, the iteration ceiling `limit`, the number of constant displacement increments that are to be applied `incs` and the magnitude of the displacement increment held in `presc`. It may be noted that the iteration ceiling does not need to be as high as when using load control. Convergence is quicker when using displacement control, especially as failure conditions are approached, since uncontrolled flow cannot occur. The 'penalty' technique is used to implement the prescribed displacements, as was used in Program 6.3.

The program follows familiar lines until the calculation of the failure function. Initially, the failure function `fnew` is obtained after adding the full elastic stress increment to those stresses existing previously. If `fnew` is positive, indicating a yielding Gauss point, then the failure function `f` is obtained using just those stresses existing previously. The scaling parameter `fac` is then calculated as described in equation (6.35). The plastic stress-strain matrix  $[\mathbf{D}^p]$  for a Mohr-Coulomb material is formed by the subroutine `mcdp1` (if implementing the von Mises criterion, the subroutine `vmdp1` should be substituted) using stresses that have been factored to ensure they lie on the failure surface. The resulting matrix `p1` is multiplied by the scaling parameter `fac` and then by the total strain increment array `eps` to yield the 'plastic' stress increment array `elseo`. This is simple 'forward Euler' integration of the 'rate' equations. 'Implicit' versions are described in the next sections. Integrals of the type described by equation (6.34) then follow, and the array `bbodylds` is accumulated from each element. It may be noted that in the algorithm presented here, the body-loads vector is completely reformed at each iteration. This is in contrast to the viscoplasticity algorithm presented in Programs 6.1–6.4, in which the body-loads vector was accumulated at each iteration.

At convergence, the stresses must be updated ready for the next displacement (load) increment. This involves adding, to the stresses remaining from the previous increment, the one-dimensional array of total stress increments `sigma` minus the one-dimensional array of corrective 'plastic' stresses `elseo`.

The example problem shown in Figure 6.23 represents a soil with strength parameters  $\phi = 30^\circ$ ,  $c = 0$  and dilation angle  $\psi = 30^\circ$ , subjected to prescribed horizontal displacements along the left face. The displacement increments are applied to the  $x$ -components of displacement at the nine nodes adjacent to the hypothetical smooth, rigid wall shown hatched. The initial stresses in the ground are calculated assuming the unit weight  $\gamma = 20 \text{ kN/m}^3$  and 'at rest' earth pressure coefficient  $K_0 = 1$ .



**Figure 6.23** Mesh and data for Program 6.5 example

Following each displacement increment, and after numerical convergence, the resultant force on the wall is calculated in two ways. Firstly, the force on the wall is computed by averaging the  $\sigma_x$  stresses at the eight Gauss points closest to the wall, and this result is held in `pav`. Secondly, the nodal reactions are back-calculated from the converged stress field as was done in Program 6.3, and this is held in `pr`. By multiplying the nodal reaction forces about their distance from the base of the wall, the overturning moment can also be estimated, held in `ot`.

The output shown in Figure 6.24 gives the step number, the accumulated wall displacement, the resultant force (averaging and reactions), the overturning moment and the number of iterations to convergence. These results are plotted in Figure 6.25 and show that the passive force builds up to a maximum value close to the Rankine solution of 30 kN/m, despite the relatively coarse mesh.

The somewhat higher result obtained by nodal reactions was also observed in the results of Program 6.3, and is due in part to the high shear stress concentration at the bottom

There are 294 equations and the skyline storage is 10521

step	disp	load(av)	load(react)	moment	iters
1	0.2000E-04	0.1097E+02	0.1197E+02	0.3678E+01	2
2	0.4000E-04	0.1194E+02	0.1311E+02	0.4023E+01	2
3	0.6000E-04	0.1292E+02	0.1425E+02	0.4368E+01	4
4	0.8000E-04	0.1385E+02	0.1535E+02	0.4676E+01	31
5	0.1000E-03	0.1477E+02	0.1644E+02	0.4971E+01	19
6	0.1200E-03	0.1569E+02	0.1752E+02	0.5262E+01	10
7	0.1400E-03	0.1660E+02	0.1860E+02	0.5548E+01	14
8	0.1600E-03	0.1751E+02	0.1968E+02	0.5833E+01	8
9	0.1800E-03	0.1842E+02	0.2076E+02	0.6115E+01	13
10	0.2000E-03	0.1933E+02	0.2183E+02	0.6397E+01	4
11	0.2200E-03	0.2021E+02	0.2288E+02	0.6656E+01	32
12	0.2400E-03	0.2107E+02	0.2392E+02	0.6904E+01	23
13	0.2600E-03	0.2193E+02	0.2495E+02	0.7148E+01	13
14	0.2800E-03	0.2279E+02	0.2597E+02	0.7384E+01	22
15	0.3000E-03	0.2363E+02	0.2698E+02	0.7614E+01	16
16	0.3200E-03	0.2444E+02	0.2796E+02	0.7819E+01	30
17	0.3400E-03	0.2523E+02	0.2884E+02	0.8004E+01	32
18	0.3600E-03	0.2601E+02	0.2970E+02	0.8186E+01	13
19	0.3800E-03	0.2675E+02	0.3048E+02	0.8355E+01	40
20	0.4000E-03	0.2744E+02	0.3120E+02	0.8519E+01	44
21	0.4200E-03	0.2810E+02	0.3187E+02	0.8676E+01	28
22	0.4400E-03	0.2866E+02	0.3243E+02	0.8815E+01	40
23	0.4600E-03	0.2918E+02	0.3293E+02	0.8945E+01	29
24	0.4800E-03	0.2961E+02	0.3330E+02	0.9068E+01	64
25	0.5000E-03	0.3001E+02	0.3363E+02	0.9180E+01	29
26	0.5200E-03	0.3029E+02	0.3387E+02	0.9264E+01	44
27	0.5400E-03	0.3043E+02	0.3403E+02	0.9313E+01	45
28	0.5600E-03	0.3056E+02	0.3416E+02	0.9353E+01	18
29	0.5800E-03	0.3067E+02	0.3427E+02	0.9387E+01	11
30	0.6000E-03	0.3076E+02	0.3437E+02	0.9414E+01	13
31	0.6200E-03	0.3085E+02	0.3447E+02	0.9439E+01	7
32	0.6400E-03	0.3092E+02	0.3455E+02	0.9459E+01	11
33	0.6600E-03	0.3099E+02	0.3463E+02	0.9474E+01	14
34	0.6800E-03	0.3105E+02	0.3468E+02	0.9486E+01	12
35	0.7000E-03	0.3110E+02	0.3474E+02	0.9497E+01	3

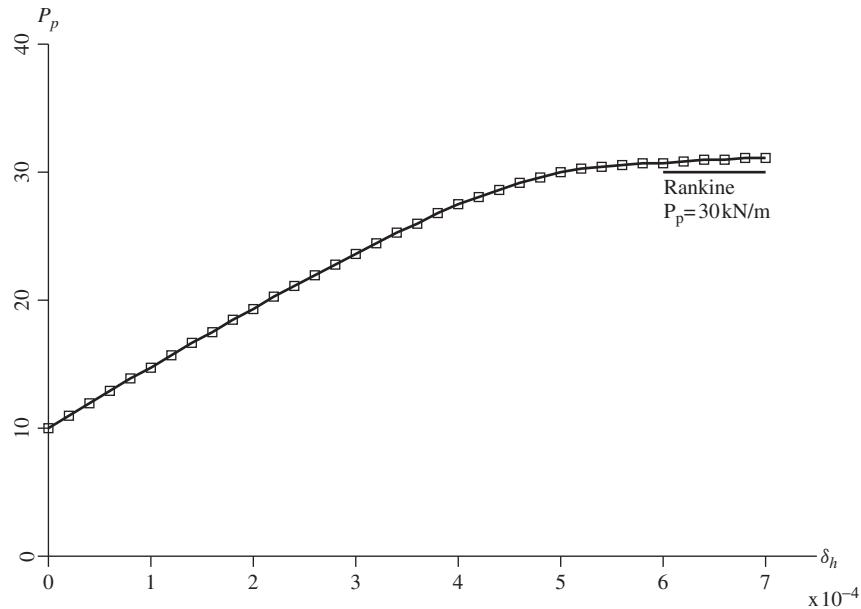
Figure 6.24 Results from Program 6.5 example

edge of the wall. The displacement vectors of the mesh corresponding to passive failure of the soil behind the wall are shown in Figure 6.26. The Rankine passive mechanism outcropping at an angle of  $30^\circ$  to the horizontal is reproduced.

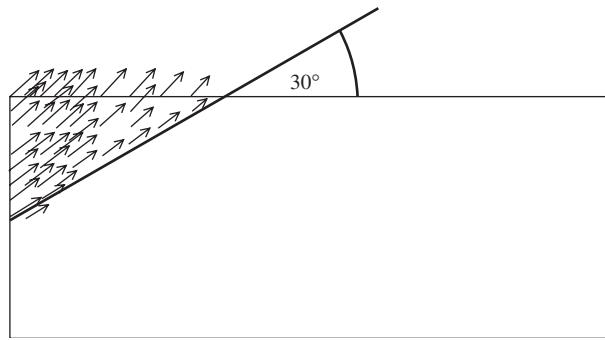
The initial stress algorithm presented in this program will tend to overestimate collapse loads, especially if the displacement (load) steps are made too big. Users are recommended to try one or two different increment sizes to test the sensitivity of the solutions. The problem is caused by incremental ‘drift’ of the stress state at individual Gauss points into illegal stress space, in spite of apparent numerical convergence. Although not included in the present work, various strategies are available (e.g., Nayak and Zienkiewicz, 1972) for drift correction. In the next section, more complicated ‘stress–return’ procedures are illustrated, which ensure stresses at each Gauss point return accurately to the yield surface.

## 6.9 Elastoplastic Rate Integration

For the purposes of this description, we return to elastic–perfectly plastic materials obeying the von Mises failure criterion. Similar, if more complicated arguments apply to Mohr–Coulomb materials.



**Figure 6.25** Passive force (based on stress averaging) vs. horizontal displacement from Program 6.5 example



**Figure 6.26** Displacement vectors at failure from Program 6.5 example

Using the notation previously developed in Sections 6.3 and 6.4, if  $F$  is the yield function and  $J_2$  the second invariant of the stress tensor, from (6.7),

$$\begin{aligned} F &= \bar{\sigma} - \sqrt{3}c_u \\ &= \sqrt{3J_2} - \sqrt{3}c_u \end{aligned} \quad (6.42)$$

where

$$J_2 = \frac{t^2}{2} = \frac{1}{6}[(\sigma_x + \sigma_y)^2 + (\sigma_y + \sigma_z)^2 + (\sigma_z + \sigma_x)^2 + 6\tau_{xy}^2 + 6\tau_{yz}^2 + 6\tau_{zx}^2] \quad (6.43)$$

The first derivative of  $F$  with respect to the stresses is

$$\left\{ \frac{\partial F}{\partial \sigma} \right\} = \{\mathbf{a}\} = \frac{\partial F}{\partial J_2} \left\{ \frac{\partial J_2}{\partial \sigma} \right\} \quad (6.44)$$

which can be written as

$$\{\mathbf{a}\} = \frac{1.5}{\sqrt{3J_2}} \begin{Bmatrix} s_x \\ s_x \\ s_x \\ 2\tau_{xy} \\ 2\tau_{yz} \\ 2\tau_{zx} \end{Bmatrix} \quad (6.45)$$

where  $s_x$ , etc. represent the deviatoric components from equations (6.3).

The second derivative of  $F$  with respect to stress is

$$\left[ \frac{\partial^2 F}{\partial \sigma^2} \right] = \left[ \frac{\partial \mathbf{a}}{\partial \sigma} \right] = \frac{1}{2\sqrt{3J_2}} [\mathbf{A}] - \frac{1}{\sqrt{3J_2}} \{\mathbf{a}\} \{\mathbf{a}\}^T \quad (6.46)$$

where

$$[\mathbf{A}] = \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \end{bmatrix} \quad (6.47)$$

Ortiz and Popov (1985) described various methods of elastoplastic rate integration, which essentially consist of an (elastic) predictor, followed by a plastic corrector to ensure the final stress is (nearly) on the yield surface.

Referring to Figure 6.27, let  $\{\sigma_X\}$  refer to the unyielded stress at the start of a step and  $\{\Delta\sigma^e\}$  the (elastic) increment. The stress crosses the yield surface at  $\{\sigma_A\}$  while the elastic increment ends up at  $\{\sigma_B\}$ . We wish to return to the ‘correct’ stresses on the yield surface at  $\{\sigma_C\}$ .

If  $\{\Delta\epsilon\}$  is the total incremental strain,  $\{\Delta\epsilon^p\}$  the incremental plastic strain and  $\lambda$  the scalar multiplier (6.31), an elastic stress-strain matrix  $[\mathbf{D}^e]$  will lead to

$$\{\sigma_C\} = \{\sigma_A\} + [\mathbf{D}^e](\{\Delta\epsilon\} - \{\Delta\epsilon^p\}) \quad (6.48)$$

where

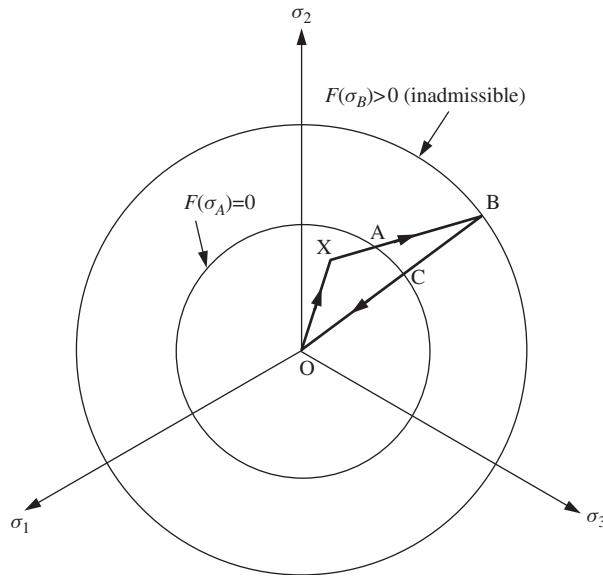
$$\{\Delta\epsilon^p\} = \lambda \{\mathbf{a}\} \quad (6.49)$$

The derivative  $\{\mathbf{a}\}$  in the above equation is evaluated at  $(1 - \beta)\{\sigma_A\} + \beta\{\sigma_C\}$ , and the scalar interpolating parameter  $\beta$  [similar to  $\theta$  in (3.99)] can vary in the range  $0 \leq \beta \leq 1$ .

### 6.9.1 Forward Euler Method

Here  $\beta = 0$  and the rate equation is integrated at the point at which the yield surface is crossed ( $\{\sigma_A\}$  in Figure 6.27). Thus

$$F(\{\sigma_X\} + \alpha\{\Delta\sigma^e\}) = 0 \quad (6.50)$$



**Figure 6.27** Stress correction

For a non-hardening von Mises material the point can be found explicitly from

$$\alpha = \frac{-3B \pm \sqrt{9B^2 - 12A(3C - 3c_u^2)}}{6A} \quad (6.51)$$

where

$$\begin{aligned} A &= \frac{1}{2}((\Delta s_x^e)^2 + (\Delta s_y^e)^2 + (\Delta s_z^e)^2) + (\Delta \tau_{xy}^e)^2 \\ B &= s_x \Delta s_x^e + s_y \Delta s_y^e + s_z \Delta s_z^e + 2\Delta \tau_{xy} \tau_{xy}^e \\ C &= \frac{1}{2}(\Delta s_x^2 + \Delta s_y^2 + \Delta s_z^2) + \Delta \tau_{xy}^2 \end{aligned} \quad (6.52)$$

but an approximate  $\alpha$  can be found by linearly interpolating between points  $X$  and  $B$  from

$$\alpha \approx \frac{-F(\{\sigma_X\})}{F(\{\sigma_B\}) - F(\{\sigma_X\})} \quad (6.53)$$

The remaining stress  $(1 - \alpha)[\mathbf{D}^e]\{\Delta \boldsymbol{\epsilon}\}$  causes the ‘illegal’ stress state outside the yield surface. For non-hardening plasticity, it is assumed that once a stress state reaches a yield surface, subsequent changes in stress may shift the stress state to a different position on the yield surface but not outside it (6.30), hence

$$\Delta F = \{\mathbf{a}\}^T \{\Delta \boldsymbol{\sigma}\} = 0 \quad (6.54)$$

thus

$$\Delta F = \{\mathbf{a}\}^T ([\mathbf{D}^e]\{\Delta \boldsymbol{\epsilon}\} - (1 - \alpha)\lambda[\mathbf{D}^e]\{\mathbf{a}\}) = 0 \quad (6.55)$$

Assuming that the derivative vector  $\{\mathbf{a}\}$  as evaluated at point  $A$  is called  $\{\mathbf{a}_A\}$ , the following expression for  $\lambda_A$ , the ‘plastic multiplier’ at  $A$ , is given by

$$\lambda_A = \frac{\{\mathbf{a}_A\}^T [\mathbf{D}^e] \{\Delta\boldsymbol{\epsilon}\}}{(1 - \alpha) \{\mathbf{a}_A\}^T [\mathbf{D}^e] \{\mathbf{a}_A\}} \quad (6.56)$$

The final stress is then

$$\{\boldsymbol{\sigma}_C\} = \{\boldsymbol{\sigma}_X\} + \{\Delta\boldsymbol{\sigma}^e\} - \lambda_A [\mathbf{D}^e] \{\mathbf{a}_A\} \quad (6.57)$$

This is the method used in equation (6.35) in which  $f = 1 - \alpha$ .

### 6.9.2 Backward Euler Method

Here the rate equation is integrated at the ‘illegal’ state  $B$  ( $\beta = 1$ ). This results in a simple evaluation of the plastic multiplier  $\lambda$  for non-hardening von Mises materials. A first-order Taylor expansion of the yield function at  $B$  gives

$$F(\{\boldsymbol{\sigma}_C\}) = F(\{\boldsymbol{\sigma}_B\}) + \left\{ \frac{\partial F}{\partial \boldsymbol{\sigma}} \right\}^T \{\Delta\boldsymbol{\sigma}\} \quad (6.58)$$

By enforcing consistency of the yield function at point  $C$ ,

$$0 = F(\{\boldsymbol{\sigma}_B\}) - \lambda_B \{\mathbf{a}_B\}^T [\mathbf{D}^e] \{\mathbf{a}_B\} \quad (6.59)$$

so that

$$\lambda_B = \frac{F(\{\boldsymbol{\sigma}_B\})}{\{\mathbf{a}_B\}^T [\mathbf{D}^e] \{\mathbf{a}_B\}} \quad (6.60)$$

The change in stress is given by

$$\{\Delta\boldsymbol{\sigma}\} = \{\Delta\boldsymbol{\sigma}^e\} - \frac{F(\{\boldsymbol{\sigma}_B\})[\mathbf{D}^e]\{\mathbf{a}_B\}}{\{\mathbf{a}_B\}^T [\mathbf{D}^e] \{\mathbf{a}_B\}} \quad (6.61)$$

and final stress by

$$\{\boldsymbol{\sigma}\} = \{\boldsymbol{\sigma}_X\} + \{\Delta\boldsymbol{\sigma}^e\} - \lambda_B [\mathbf{D}^e] \{\mathbf{a}_B\} \quad (6.62)$$

Rice and Tracey (1973) advocated a mean normal method for a von Mises yield criterion so that

$$\frac{\{\mathbf{a}_A + \mathbf{a}_B\}^T \{\Delta\boldsymbol{\sigma}^e\}}{2} = 0 \quad (6.63)$$

In a von Mises yield criterion under plane strain and 3D stress states, the yield surface appears as a circle on the deviatoric plane. Any ‘illegal’ stress can be corrected along a radial path directed from the hydrostatic stress axis. The final deviatoric stress at point  $C$  is

$$\{\mathbf{s}_C\} = \frac{\sqrt{3c_u}}{\sqrt{3J_2}} \{\mathbf{s}_B\} \quad (6.64)$$

and the components of  $\{\boldsymbol{\sigma}_C\}$  can then be determined by superimposing the hydrostatic stress from point  $B$ .

In practice, it has been found that this method offers no advantages over forward Euler in constant stiffness algorithms. The same is not true for tangent stiffness methods, as is shown in the next section.

## 6.10 Tangent Stiffness Approaches

The difference between constant stiffness and tangent stiffness methods was discussed in Section 6.1. In general, constant stiffness methods can be attractive in displacement-controlled situations (see, e.g., Figure 6.24 where the number of iterations per displacement increment is modest) but in load-controlled situations, particularly close to collapse, large numbers of iterations tend to arise (see, e.g., Figure 6.20). Tangent stiffness approaches require fewer iterations per load step, however this saving is counterbalanced by the speed of constant stiffness methods, in which the global stiffness matrix is only factorised once. Tangent stiffness methods, with backward Euler integration, can significantly improve the convergence properties of algorithms to the point where the cost of re-forming and re-factorising the global stiffness can be justified.

### 6.10.1 Inconsistent Tangent Matrix

The change in stress is composed of two parts, the elastic predictor  $[\mathbf{D}^e]\{\Delta\epsilon\}$  and a plastic corrector  $\lambda[\mathbf{D}^e]\{\mathbf{a}\}$ , that is

$$\{\Delta\sigma\} = [\mathbf{D}^e](\{\Delta\epsilon\} - \lambda\{\mathbf{a}\}) \quad (6.65)$$

Substituting  $\lambda$  from (6.56) ( $\alpha = 0$ ) into the above equation gives

$$\{\Delta\sigma\} = [\mathbf{D}^e] \left( \{\Delta\epsilon\} - \frac{\{\mathbf{a}\}^T [\mathbf{D}^e]\{\Delta\epsilon\}}{\{\mathbf{a}\}^T [\mathbf{D}^e]\{\mathbf{a}\}} \{\mathbf{a}\} \right) \quad (6.66)$$

and hence

$$\{\Delta\sigma\} = \left( [\mathbf{D}^e] - \frac{[\mathbf{D}^e]\{\mathbf{a}\}\{\mathbf{a}\}^T [\mathbf{D}^e]}{\{\mathbf{a}\}^T [\mathbf{D}^e]\{\mathbf{a}\}} \right) \{\Delta\epsilon\} \quad (6.67)$$

or

$$\{\Delta\sigma\} = [\mathbf{D}^{ep}]\{\Delta\epsilon\} \quad (6.68)$$

where  $[\mathbf{D}^{ep}]$  is known as the standard or ‘inconsistent’ tangent matrix.

### 6.10.2 Consistent Tangent Matrix

With the backward Euler integration scheme, a consistent tangent matrix can be formed:

$$\begin{aligned} \{\sigma\} &= \{\sigma_B\} - \lambda_B [\mathbf{D}^e] \{\mathbf{a}_B\} \\ &= (\{\sigma_X\} + [\mathbf{D}^e]\{\Delta\epsilon\}) - \lambda_B [\mathbf{D}^e] \{\mathbf{a}_B\} \end{aligned} \quad (6.69)$$

On differentiation, we get

$$\{\Delta\sigma\} = [\mathbf{D}^e]\{\Delta\epsilon\} - \Delta\lambda[\mathbf{D}^e]\{\mathbf{a}\} - \lambda_B[\mathbf{D}^e] \left[ \left( \frac{\partial \mathbf{a}}{\partial \sigma} \right)_B \right] \{\Delta\sigma\} \quad (6.70)$$

or

$$\{\Delta\sigma\} = \left[ [\mathbf{I}] + \lambda_B[\mathbf{D}^e] \left[ \left( \frac{\partial \mathbf{a}}{\partial \sigma} \right)_B \right] \right]^{-1} [\mathbf{D}^e](\{\Delta\epsilon\} - \Delta\lambda\{\mathbf{a}\}) \quad (6.71)$$

$$= [\mathbf{R}](\{\Delta\epsilon\} - \Delta\lambda\{\mathbf{a}\}) \quad (6.72)$$

and hence

$$\{\Delta\sigma\} = \left( [\mathbf{R}] - \frac{[\mathbf{R}]\{\mathbf{a}\}\{\mathbf{a}\}^T[\mathbf{R}]}{\{\mathbf{a}\}^T[\mathbf{R}]\{\mathbf{a}\}} \right) \{\Delta\epsilon\} \quad (6.73)$$

or

$$\{\Delta\sigma\} = [\mathbf{D}^{epc}]\{\Delta\epsilon\} \quad (6.74)$$

where  $[\mathbf{D}^{epc}]$  is known as the ‘consistent’ tangent matrix.

### 6.10.3 Convergence Criterion

Programs 6.1 to 6.5 used a simple convergence criterion that was essentially based on the amount that the nodal displacements changed from one iteration to the next. Convergence was said to have occurred, if the absolute change in all the displacement components (often called loads in the programs), as a fraction of the maximum absolute component, was less than a tolerance `tol` of, say, 0.001.

When the convergence of loads is examined, it is found that in typical problems nearly all Gauss points converge early, and most of the time is taken in forcing the stresses at a few points towards the yield surface.

In the consistent tangent method as described in the next three programs, the extra work done in reassembly is such that all Gauss points converge much faster to the yield surface. Both local `ltol` and global `tol` convergence criteria are needed. In contrast, there can be no concept of a converging `bdflds`. Rather, the residual remaining in `bdflds` tends to zero as convergence is approached (this is actually how many codes operate for the constant stiffness, forward Euler, case as well). A criterion based on the size of the maximum component of the reducing `bdflds` as a percentage of, say, the L2-norm of `bdflds` can be used.

In practice, when an element assembly technique is chosen, the strategy for constant stiffness is simple to implement and may be just as efficient computationally as the consistent tangent approach. This is because the plastic iterations involve only forward and back-substitutions in a direct equation-solving process.

When a tangent stiffness method is used, the extra time involved in reforming the stiffness matrices and completely resolving the equilibrium equations can more than compensate for the reduced iteration counts.

However, when iterative strategies are adopted (e.g., Program 6.2), the equilibrium equations have to be ‘reassembled’ and solved on every iteration anyway. In these circumstances, the consistent tangent stiffness with backward Euler return, leading to low iteration counts, is essential.

**Program 6.6 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)**

```

PROGRAM p66
!-----
! 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).
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,incs,iter,iy,k,limit,loaded_nodes,ndim=2,ndof=16,nels,    &
neq,nip=4,nlen,nn,nod=8,nodof=2,nprops=3,np_types,nst=4,nxe,nye
REAL(iwp)::bot,det,diam,dsbar,dslam,d3=3.0_iwp,end_time,fnew,lode_theta,  &
ltol,one=1.0_iwp,ptot,sigm,start_time,tol,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:),g_g(:,:,),g_num(:,:,),kdiag(:,nf(:,:,), &
node(:,num(:)
REAL(iwp),ALLOCATABLE::acat(:,be(:,bdylds(:,caflow(:,&
coord(:,daatd(:,ddylds(:,dee(:,der(:,deriv(:,dload(:,&
dsigma(:,eld(:,eload(:,eps(:,g_coord(:,jac(:,km(:,kv(:,&
loads(:,loads(:,points(:,prop(:,qinc(:,qinva(:,qinvr(:,&
qmat(:,ress(:,rmat(:,sigma(:,stress(:,tensor(:, :,&
tensorl(:, :,totd(:,totdl(:,trial(:,val(:,vmfl(:,vmfla(:,&
vmflq(:,weights(:,x_coords(:,y_coords(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
CALL CPU_TIME(start_time); READ(10,*)nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),    &
x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),    &
tensor(nst,nip,nels),g_g(ndof,nels),coord(nod,ndim),jac(ndim,ndim),    &
der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),    &
km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),eload(ndof),g(ndof),    &
vmfl(nst),qinva(nst),stress(nst),dload(ndof),caflow(nst),dsigma(nst),    &
ress(nst),rmat(nst,nst),acat(nst,nst),qmat(nst,nst),qinva(nst),    &
daatd(nst,nst),vmflq(nst),vmfla(nst),prop(nprefs,np_types),etype(nels),    &
tensorl(nst,nip,nels),trial(nst))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords
CALL bc_rect(nxe,nye,nf,'y'); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),totd(0:neq),ddylds(0:neq), &
loads(0:neq),totdl(0:neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); CALL fkdiag(kdiag,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))

```

```

WRITE(11, '(2(A,I7))')
  &
  " There are",neq," equations and the skyline storage is",kdiag(neq)
kv=zero; totd=zero; tensor=zero; tensorl=zero
CALL sample(element,points,weights)
!-----starting element stiffness integration and assembly-----
elements_2: DO iel=1,nels
  CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
  num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
  gauss_pts_1: DO i=1,nip
    CALL shape_der(der,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
  END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read load weightings and factorise equations-----
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes); CALL sparin(kv,kdiag)
!-----
READ(10,*)ltol,tol,limit,incs; ALLOCATE(qinc(incs))
READ(10,*)qinc; ptot=zero
WRITE(11,'(/A)')" step      load          disp        iters"
load_increments: DO iy=1,incs
  ptot=ptot+qinc(iy); totdl=zero; loads=zero; loadsr=zero
  bdylds=zero; iters=0
!-----load increment loop-----
DO i=1,loaded_nodes
  loads(nf(:,node(i)))=val(i,:)*qinc(iy) ! load increment
  loadsr(nf(:,node(i)))=val(i,:)*ptot ! total load
END DO
!-----plastic iteration loop-----
iterations: DO
  iters=iters+1
  WRITE(*,'(A,F8.2,A,I4)')" load",ptot," iteration",iters
  IF(iters/=1)loads=zero; loads=loads+bdylds
  CALL spabac(kv,loads,kdiag); loads(0)=zero; totdl=totdl+loads
  ddylds=zero; kv=zero
!-----go round the Gauss Points -----
elements_3: DO iel=1,nels
  dload=zero; num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
  g=g_g(:,iel); km=zero; eld=totdl(g)
  gauss_pts_2: DO i=1,nip
    CALL shape_der(der,points,i); jac=MATMUL(der,coord)
    det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
    CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
    CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
    sigma=MATMUL(dee,eps); stress=sigma+tensorl(:,i,iel)
    trial=stress; CALL invar(stress,sigm,dsbar,lode_theta)
!-----check whether yield is violated-----
  fnew=dsbar-SQRT(d3)*prop(1,etype(iel))
  IF(fnew>=zero)THEN
    dlam=zero
    iterate_on_fnew: DO
      CALL vmf1(stress,dsbar,vmf1); caflow=MATMUL(dee,vmf1)
      ress=stress-trial+caflow*dlam; CALL fmacat(vmf1,acat)
      acat=acat/dsbar; qmat=dlam*MATMUL(dee,acat)
      DO k=1,4; qmat(k,k)=qmat(k,k)+one; END DO
      CALL invert(qmat); vmflq=MATMUL(vmf1,qmat)
    END IF
  END DO
END DO

```

```

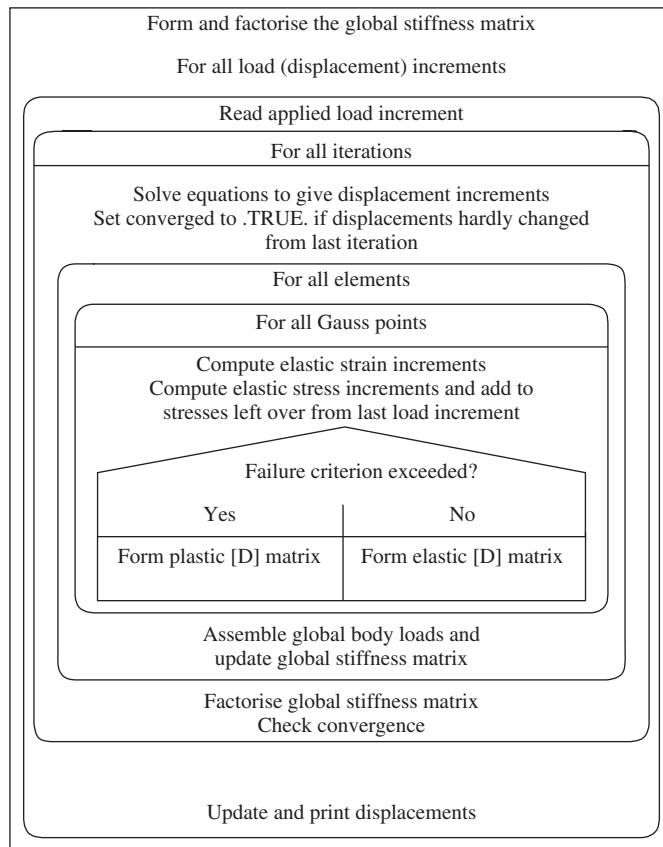
vmfla=MATMUL(vmflq,dee)
dslam=(fnew-DOT_PRODUCT(vmflq,ress))/DOT_PRODUCT(vmfla,vmfl)
dsigma=-MATMUL(qmat,ress)-MATMUL(MATMUL(qmat,dee),vmfl)*dsla
stress=stress+dsigma; CALL invar(stress,sigm,dsbar,lode_theta)
fnew=dsbar-SQRT(d3)*prop(1,etype(iel)); dlam=dlam+dsla
IF(fnew<ltol)EXIT
END DO iterate_on_fnew; CALL vmflow(stress,dsbar,vmfl)
CALL fmrrmat(vmfl,dsbar,dlam,dee,rmat); caflow=MATMUL(rmat,vmfl)
bot=DOT_PRODUCT(vmfl,caflow); CALL formaa(vmfl,rmat,daatd)
dee=rmat-daatd/bot
END IF
km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----update the Gauss Point stresses-----
tensor(:,i,iel)=stress; eload=MATMUL(stress,bee)
dload=dload+eload*det*weights(i)
END DO gauss_pts_2
!-----compute the total bodyloads vector-----
ddylds(g)=ddylds(g)+dload; ddylds(0)=zero; CALL fsparv(kv,km,g,kdiag)
END DO elements_3
CALL sparin(kv,kdiag); bdylds=loads_r-ddylds; bdylds(0)=zero
IF(norm(bdylds(1:))/norm(loads_r(1:))<tol)converged=.TRUE.
IF(iters==1)converged=.FALSE.; IF(converged.OR.iters==limit)EXIT
END DO iterations
tensor=tensor; totd=totd+totd
WRITE(11,'(I5,2E12.4,I5)')iy,ptot,totd(nf(2,node(1))),iters
IF(iters==limit)EXIT
END DO load_increments
CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(totd,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
CALL CPU_TIME(end_time)
WRITE(11,'(A,F12.4)')" Time taken = ",end_time-start_time
STOP
END PROGRAM p66

```

The structure chart for a typical tangent stiffness approach is shown in Figure 6.28. Preliminary element loops, `elements_1` and `elements_2`, set up the geometry and the starting tangent matrix, respectively. The load increment loop is entered and new subroutines encountered are `vmflow`, which forms the von Mises flow vector from equation (6.45) and `fmrrmat`, `formaa` and `fmacat`, which are used in the development of equation (6.73) at every Gauss point in the `elements_3` loop. The  $4 \times 4$  matrix `qmat` has to be inverted to complete the return strategy at each Gauss point and ultimately the consistent tangent `dee` matrix is obtained, leading to the consistent `km`.

This program demonstrates a well-established algorithm of the consistent tangent stiffness type, sometimes called the closest point projection method, for a von Mises failure criterion. The algorithm is quite general, and has been discussed in detail elsewhere (e.g., Simo and Taylor, 1985; Belytchko *et al.*, 2000; Huang and Griffiths, 2008, 2009). As usual, loads are applied incrementally, however unlike previous programs in this chapter, the global stiffness matrix is reassembled and factorised at every iteration. Although this involves considerably more work at each iteration, the number of iterations for each load step is significantly reduced because the tangent operator is more accurately predicting the trajectory of the stress-strain curve as shown in Figure 6.2.

Figure 6.29 gives the data for the problem analysed, which is the same as that considered at the beginning of this chapter and shown in Figure 6.9. Boundary condition data has



**Figure 6.28** Structure chart for variable stiffness algorithm with assembly

```

nxe  nye  np_types
8      4      1

prop (c_u,e,v)
100.0  1.0E5  0.3

etype (not needed)

x_coords, y_coords
0.0   1.0   2.0   3.0   4.0   5.5   7.0   9.0   12.0
0.0  -1.25 -2.5  -3.75 -5.0

loaded_nodes, (node(i),val(i,:),i=1,loaded_nodes)
5
1    0.0   -0.166667     10   0.0   -0.666667     15   0.0   -0.333333
24   0.0   -0.666667    29   0.0   -0.166667

ltol      tol      limit
0.00005  0.001    25

incs, (qinc(i),i=1,incs)
10
200.0 100.0 50.0 50.0 50.0 30.0 20.0 10.0 5.0 4.0
  
```

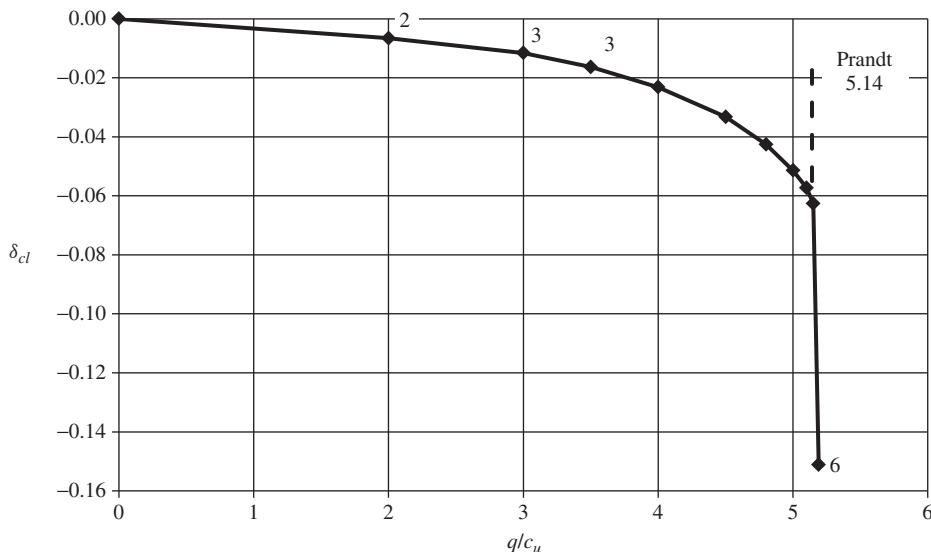
**Figure 6.29** Data for Program 6.6 example

There are 192 equations and the skyline storage is 4322

step	load	disp	iters
1	0.2000E+03	-0.6593E-02	2
2	0.3000E+03	-0.1161E-01	3
3	0.3500E+03	-0.1635E-01	3
4	0.4000E+03	-0.2319E-01	3
5	0.4500E+03	-0.3326E-01	3
6	0.4800E+03	-0.4259E-01	3
7	0.5000E+03	-0.5140E-01	3
8	0.5100E+03	-0.5730E-01	3
9	0.5150E+03	-0.6265E-01	3
10	0.5190E+03	-0.1511E+00	6

Time taken = 0.0936

**Figure 6.30** Results from Program 6.6 example



**Figure 6.31** Bearing stress vs. centreline displacement from Program 6.6 example

been provided by subroutine `bc_rect`, and this will also be the case with Programs 6.7 and 6.8 to follow. The only new information required in this case is an additional local tolerance `ltol=0.00005` which applies to iterations performed at the Gauss point level, while the iteration ceiling `limit` can be assigned a much more economical value of, say, 25.

The results are listed as Figure 6.30 and are found to be very similar to those in Figure 6.10, but with significantly fewer iterations. On the final increment (at ‘failure’), Program 6.6 took only six iterations compared to Program 6.1 which took (at least) 250. The computed load–displacement curve is shown in Figure 6.31.

**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**

```

PROGRAM p67
!-----
! Program 6.7 Plane strain bearing capacity analysis of an elastic-plastic
! (von Mises) material using 8-node rectangular quadrilaterals.
! Consistent tangent stiffness. Closest Point Projection
! Method (CPPM). No global stiffness matrix assembly.
! Diagonally preconditioned conjugate gradient solver.
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,cg_tot,i,iel,incs,iters,iy,k,limit,
loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nlen,nn,nod=8,nodof=2,
nprops=3,np_types,nst=4,nxe,nye
REAL(iwp)::alpha,beta,bot,cg_tol,det,dlam,dsbar,dslam,d3=3.0_iwp,
end_time,fnew,lode_theta,ltol,one=1.0_iwp,ptot,sigm,start_time,tol,up,
zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
LOGICAL::cg_converged,converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,),g(:,),g_g(:,:,),g_num(:,:,),nf(:,:,),node(:), &
num(:)
REAL(iwp),ALLOCATABLE::acat(:,:,),bee(:,:,),bdylds(:,:,),caflow(:,:,),coord(:,:),
d(:,:,),daatd(:,:,),ddylds(:,:,),dee(:,:,),der(:,:,),deriv(:,:,),diag_precon(:,:,),
dl(:,:,),dload(:,:,),dsigma(:,:,),eld(:,:,),eload(:,:,),eps(:,:,),g_coord(:,:,),
jac(:,:,),km(:,:,),kv(:,:,),loads(:,:,),loadsrs(:,:,),p(:,:,),points(:,:,),prop(:,:,),
qinc(:,:,),qinva(:,:,),qinvr(:,:,),qmat(:,:,),ress(:,:,),rmat(:,:,),trial(:),
sigma(:,:,),storkm(:,:,,:),stress(:,:,),tensor(:,:,,:),tensorl(:,:,,:),totd(:,:,),
totdl(:,:,),u(:,:,),val(:,:,),vmfl(:,:,),vmfla(:,:,),vmflq(:,:,),weights(:,x(:), &
xnew(:,x_coords(:,y_coords(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
CALL CPU_TIME(start_time); READ(10,*)nxe,nye,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst), &
tensor(nst,nip,nels),g_g(ndof,nels),coord(nod,ndim),jac(ndim,ndim), &
der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof), &
km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),eload(ndof),g(ndof), &
vmfl(nst),qinvr(nst),stress(nst),dload(ndof),caflow(nst),dsigma(nst), &
ress(nst),rmat(nst,nst),acat(nst,nst),qmat(nst,nst),qinva(nst), &
daatd(nst,nst),vmflq(nst),vmfla(nst),prop(nprops,np_types),etype(nels), &
tensorl(nst,nip,nels),trial(nst),storkm(ndof,ndof,nels),dl(nip,nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords; CALL bc_rect(nxe,nye,nf,'y'); neq=MAXVAL(nf)
ALLOCATE(loads(0:neq),bdylds(0:neq),totd(0:neq),ddylds(0:neq), &

```

```

loadsr(0:neq),totdl(0:neq),p(0:neq),xnew(0:neq),x(0:neq),          &
diag_precon(0:neq),d(0:neq),u(0:neq)
!-----loop to set up global arrays-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
WRITE(11,'(A,I7,A)')" There are",neq," equations"
WRITE(11,'(/A)')
" step      load      disp      iters      cg iters/plastic iter"
totd=zero; p=zero; xnew=zero; tensor=zero; tensorl=zero; dl=zero
diag_precon=zero; CALL sample(element,points,weights)
!-----starting element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i);
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1; storkm(:,:,iel)=km
    DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
END DO elements_2
diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
!-----read load weightings-----
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
!-----load increment loop-----
READ(10,*)ltol,tol,limit,incs; ALLOCATE(qinc(incs))
READ(10,*)qinc; ptot=zero
load_increments: DO iy=1,incs
    ptot=ptot+qinc(iy)
    totdl=zero; loads=zero; bdylds=zero; cg_tot=0; iters=0
!-----load increment loop-----
    DO i=1,loaded_nodes
        loads(nf(:,node(i)))=val(i,:)*qinc(iy) ! load increment
        loadsr(nf(:,node(i)))=val(i,:)*ptot ! total load
    END DO
!-----plastic iteration loop-----
    iterations: DO
        iters=iters+1
        WRITE(*,'(A,F8.2,A,I4)')" load",ptot," iteration",iters
        IF(iters/=1)loads=zero; loads=loads+bdylds; d=diag_precon*loads; p=d
!-----pcg equation solution-----
        x=zero; cg_iters=0
        pcg: DO
            cg_iters=cg_iters+1
            u=zero
            elements_3: DO iel=1,nels
                g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
            END DO elements_3
            up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
            loads=loads-u*alpha; d=diag_precon*loads
        END DO pcg
    END DO iterations
END DO load_increments

```

```

beta=DOT_PRODUCT.loads,d)/up; p=d+p*beta
call checon(xnew,x,cg_tol,cg_converged)
IF(cg_converged.OR.cg_iters==cg_limit)EXIT
END DO pcg
cg_tot=cg_tot+cg_iters; loads=xnew; loads(0)=zero; totdl=totdl+loads
ddylds=zero; diag_precon=zero
!-----go round the Gauss Points -----
elements_4: DO iel=1,nels
  dload=zero; num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
  g=g(:,iel); km=zero; eld=totdl(g)
  gauss_pts_2: DO i=1,nip
    CALL shape_der(der,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
    CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
    sigma=MATMUL(dee,eps); stress=sigma+tensorl(:,i,i,iel)
    trial=stress; CALL invar(stress,sigm,dsbar,lode_theta)
  !-----check whether yield is violated-----
  fnew=dsbar-SQRT(d3)*prop(1,etype(iel))
  IF(fnew>=zero)THEN
    dlam=zero
    iterate_on_fnew: DO
      CALL vmflow(stress,dsbar,vmfl); caflow=MATMUL(dee,vmfl)
      ress=stress-trial+caflow*dlam; CALL fmacat(vmfl,acat)
      acat=acat/dsbar; qmat=dlam*MATMUL(dee,acat)
      DO k=1,4; qmat(k,k)=qmat(k,k)+one; END DO
      CALL invert(qmat); vmflq=MATMUL(vmfl,qmat)
      vmfla=MATMUL(vmflq,dee)
      dslam=(fnew-DOT_PRODUCT(vmflq,ress))/DOT_PRODUCT(vmfla,vmfl)
      dsigma=-MATMUL(qmat,ress)-MATMUL(MATMUL(qmat,dee),vmfl)*dsla
      stress=stress+dsigma; CALL invar(stress,sigm,dsbar,lode_theta)
      fnew=dsbar-SQRT(d3)*prop(1,etype(iel)); dlam=dlam+dsla
      IF(fnew<ltol)EXIT
    END DO iterate_on_fnew
    CALL vmflow(stress,dsbar,vmfl)
    CALL fmrrmat(vmfl,dsbar,dlam,dee,rmat); caflow=MATMUL(rmat,vmfl)
    bot=DOT_PRODUCT(vmfl,caflow); CALL formaa(vmfl,rmat,daatd)
    dee=rmat-daatd/bot
  END IF
  km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----update the Gauss Point stresses-----
  tensor(:,i,iel)=stress; eload=MATMUL(stress,bee)
  dload=dload+eload*det*weights(i)
END DO gauss_pts_2
!-----compute the total bodyloads vector-----
  ddylds(g)=ddylds(g)+dload; ddylds(0)=zero; storkm(:,:,iel)=km
  DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
END DO elements_4
diag_precon(1:neq)=one/diag_precon(1:neq); diag_precon(0)=zero
bdylds=loadsrd-ddylds; bdylds(0)=zero
IF(norm(bdylds(1:))/norm(loadsrd(1:))<tol)converged=.TRUE.
IF(iters==1)converged=.FALSE.; IF(converged.OR.iters==limit)EXIT
END DO iterations
tensorl=tensor; totd=totd+totdl
WRITE(11,'(I5,2E12.4,I5,F17.2)')
  iy,ptot,totd(nf(2,node(1))),iters,REAL(cg_tot)/REAL(iters)
  &

```

```

    IF(iters==limit)EXIT
END DO load_increments
CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(totd,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
CALL CPU_TIME(end_time)
WRITE(11,'(A,F12.4)')" Time taken = ",end_time-start_time
STOP
END PROGRAM p67

```

The ‘element-by-element’ version of the constant stiffness method, Program 6.2, was inefficient (at least on a scalar computer) due to the large number of repeated equation solutions. Program 6.6 has shown that a tangent stiffness approach, with consistent return, has reduced the number of equation solutions to three for most of the load increments. With significantly fewer equations to be solved, there is less benefit to be gained from factorisation, implying that an ‘element-by-element’ approach using an iterative preconditioned conjugate gradient solver could be appropriate. Program 6.7 should be seen as a merging of Programs 6.2 and 6.6. The data for the problem to be solved, again the original one of Figure 6.9, are shown in Figure 6.32. Extra information, as compared with the data for Program 6.6, is limited to the conjugate gradient tolerance and iteration limit, set respectively to  $cg\_tol=0.0001$  and  $cg\_limit=150$ .

The output is listed as Figure 6.33, which can be compared with Figure 6.30. Between 54 and 143 conjugate gradient iterations per plastic iteration were required, but the program ran faster than Program 6.1 for the solution of this problem, even in scalar mode. In parallel implementations, there is a trade-off between constant stiffness and tangent stiffness methods because for the latter, all yielded elements are different, although their geometries and elastic properties may be identical.

```

nxe  nye  cg_tol   cg_limit  np_types
8      4     0.0001      150        1

prop (c_u,e,v)
100.0  1.0E5  0.3

etype (not needed)

x_coords, y_coords
0.0    1.0    2.0    3.0    4.0    5.5    7.0    9.0   12.0
0.0   -1.25   -2.5   -3.75   -5.0

loaded_nodes, (node(i),val(i,:),i=1,loaded_nodes)
5
1     0.0   -0.166667      10    0.0   -0.666667      15    0.0   -0.333333
24    0.0   -0.666667      29    0.0   -0.166667

ltol      tol      limit
0.00005  0.001    25

incs,(qinc(i),i=1,incs)
10
200.0 100.0 50.0 50.0 50.0 30.0 20.0 10.0 5.0 4.0

```

**Figure 6.32** Data for Program 6.7 example

```

There are      192 equations

step    load       disp     iters     cg iters/plastic iter
 1  0.2000E+03 -0.6593E-02   2        54.00
 2  0.3000E+03 -0.1161E-01   3        54.67
 3  0.3500E+03 -0.1635E-01   3        56.67
 4  0.4000E+03 -0.2319E-01   3        63.00
 5  0.4500E+03 -0.3326E-01   3        70.67
 6  0.4800E+03 -0.4259E-01   3        76.00
 7  0.5000E+03 -0.5140E-01   3        83.67
 8  0.5100E+03 -0.5730E-01   3        87.00
 9  0.5150E+03 -0.6265E-01   3        93.67
10  0.5190E+03 -0.1457E+00  11       142.73
Time taken =          0.3744

```

**Figure 6.33** Results from Program 6.7 example

**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’**

```

PROGRAM P68
!-----
! Program p68 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). Line search technique.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,j,iel,incs,jj,iters,iy,limit,loaded_nodes,ndim=2,ndof=16,nels,&
neq,nip=4,nlen,nn,nod=8,nodof=2,nprops=3,np_types,nst=4,nxe,nye
REAL(iwp)::bot,con1,con2,det,dsbar,d2=2.0_iwp,d3=3.0_iwp,end_time,fnew, &
length,lode_theta,one=1.0_iwp,ptot,pt5=0.5_iwp,sigm,start_time,sx,sy, &
sz,s0,s1,tol,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:, ),g(:, ),g_g(:, :, ),g_num(:, :, ),kdiag(:, ),nf(:, :, ), &
node(:, ),num(:, )
REAL(iwp),ALLOCATABLE::bee(:, :, ),bdylds(:, ),bdylds0(:, ),coord(:, :, ), &
ddyls(:, ),dee(:, :, ),deep(:, :, ),der(:, :, ),deriv(:, :, ),dload(:, ),eld(:, ), &
eload(:, ),eps(:, ),g_coord(:, :, ),jac(:, :, ),km(:, :, ),kv(:, ),loads(:, ),loadsrs(:, ),&
points(:, :, ),prop(:, :, ),qinc(:, ),sigma(:, ),stress(:, ),tensor(:, :, :, ), &
tensorl(:, :, :, ),totd(:, ),totdl(:, ),totdll(:, ),totdlo(:, ),val(:, :, ), &
weights(:, ),x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
CALL cpu_time(start_time); READ(10,*)nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
x_coords(nxe+1),y_coords(nye+1),num(nod),dee(nst,nst),deep(nst,nst), &
dload(ndof),tensor(nst,nip,nels),g_g(ndof,nels),coord(nod,ndim), &

```

```

jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),          &
bee(nst,ndof),km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),eload(ndof), &
g(ndof),stress(nst),prop(nprops,np_types),etype(nels),                  &
tensorl(nst,nip,nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords; CALL bc_rect(nxe,nye,nf,'y'); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),totd(0:neq),ddylds(0:neq),&
 loadsr(0:neq),totdl(0:neq),totdlo(0:neq),bdylds0(0:neq),totdll(0:neq))
!-----loop the elements to find global arrays sizes-----
kdiag=0
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); CALL fkdiag(kdiag,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I7))')                                     &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
kv=zero; totd=zero; tensor=zero; tensorl=zero
CALL sample(element,points,weights)
!-----starting element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat (bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read load weightings and factorise equations-----
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes); CALL sparin(kv,kdiag)
!-----load increment loop-----
READ(10,*)tol,limit,incs; ALLOCATE(qinc(incs))
READ(10,*)qinc; ptot=zero; bdylds=zero
WRITE(11,'(/A)')" step      load      disp      iters"
load_increments: DO iy=1,incs
    ptot=ptot+qinc(iy); bdylds=zero; loads=zero; loadsr=zero; totdl=zero
    DO i=1,loaded_nodes
        loads(nf(:,node(i)))=val(i,:)*qinc(iy)
        loadsr(nf(:,node(i)))=val(i,:)*ptot
    END DO; iters=0
!-----plastic iteration loop-----
iterations: DO
    iters=iters+1
    WRITE(*,'(A,F8.2,A,I4)')" load",ptot," iteration",iters
    IF(iters/=1)loads=zero; loads=loads+bdylds; bdylds0=loads
    CALL spabac(kv,loads,kdiag); loads(0)=zero; length=one; totdlo=loads
    totdll=totdl; s0=DOT_PRODUCT(bdylds0,totdlo)
    line_search: DO jj=1,2
        totdl=totdll+length*totdlo; bdylds=zero; ddylds=zero; kv=zero
    !-----go round the Gauss Points -----
    elements_3: DO iel=1,nels
        dload=zero; num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
        g=g_g(:,iel); km=zero; eld=totdl(g)

```

```

CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
bot=prop(2,etype(iel))/(one+prop(3,etype(iel)))
gauss_pts_2: DO i=1,nip
    CALL shape_der(der,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    eps=MATMUL(bee,eld); sigma=MATMUL(dee,eps)
    stress=sigma+tensorl(:,i,iel)
    CALL invar(stress,sigm,dsbar,lode_theta)
!-----check whether yield is violated-----
fnew=dsbar-SQRT(d3)*prop(1,etype(iel)); deep=dee
IF(fnew>=zero)THEN
    con1=fnew/dsbar; con2=d3/d2*(one-con1)/dsbar**2*bot
    sx=stress(1)-sigm; sy=stress(2)-sigm; sz=stress(4)-sigm
    deep(1,1)=dee(1,1)-con2*sx*sx-con1*d2/d3*bot
    deep(1,2)=dee(1,2)-con2*sx*sy+con1/d3*bot
    deep(1,3)=dee(1,3)-con2*stress(3)*sx
    deep(1,4)=dee(1,4)-con2*sx*sz+con1/d3*bot
    deep(2,2)=dee(2,2)-con2*sy*sy-con1*d2/d3*bot
    deep(2,3)=dee(2,3)-con2*stress(3)*sy
    deep(2,4)=dee(2,4)-con2*sy*sz+con1/d3*bot
    deep(3,3)=dee(3,3)-con2*stress(3)**2-con1/d2*bot
    deep(3,4)=dee(3,4)-con2*stress(3)*sz
    deep(4,4)=dee(4,4)-con2*sz*sz-con1*d2/d3*bot
    DO j=1,3; deep(j+1:4,j)=deep(j,j+1:4); END DO
    stress(1)=stress(1)-con1*sx; stress(2)=stress(2)-con1*sy
    stress(4)=stress(4)-con1*sz; stress(3)=stress(3)-con1*stress(3)
END IF
km=km+MATMUL(MATMUL(TRANSPOSE(bee),deep),bee)*det*weights(i)
!-----update the Gauss Point stresses-----
tensor(:,i,iel)=stress; eload=MATMUL(stress,bee)
dload=dload+eload*det*weights(i)
END DO gauss_pts_2
!-----compute the total bodyloads vector-----
ddylds(g)=ddylds(g)+dload; ddylds(0)=zero
CALL fsparv(kv,km,g,kdiag)
END DO elements_3
CALL sparin(kv,kdiag); bdylds=loadsrt-dylds; bdylds(0)=zero
s1=DOT_PRODUCT(bdylds,totdlo); IF(ABS(s1)<=pt5*ABS(s0))EXIT
IF(s0/s1>zero)THEN; length=pt5*s0/s1
ELSE; length=s0/s1*pt5+SQRT((s0/s1*pt5)**2-s0/s1); ENDIF
END DO line_search
IF(iters==1)converged=.FALSE.
IF(iters/=1.AND.norm(bdylds(1:))/norm(loadsrt(1:))<tol)converged=.TRUE.
IF(converged.OR.iters==limit)EXIT
END DO iterations
tensor=tensor; totd=totd+totd
WRITE(11,'(I5,2E12.4,I5)')iy,ptot,totd(nf(2,node(1))),iters
IF(iters==limit)EXIT
END DO load_increments
CALL dismsh(totd,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(totd,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
CALL cpu_time(end_time)
WRITE(11,'(A,F12.4)')" Time taken = ",end_time-start_time
STOP
END PROGRAM p68

```

```

nxe  nye  np_types
8      4      1

prop (cu,e,v)
100.0  1.0E5  0.3

etype (not needed)

x_coords, y_coords
0.0   1.0    2.0    3.0    4.0    5.5    7.0    9.0   12.0
0.0  -1.25   -2.5   -3.75  -5.0

loaded_nodes, (node(i),val(i,:),i=1,loaded_nodes)
5
1    0.0   -0.166667     10   0.0   -0.666667     15   0.0   -0.333333
24   0.0   -0.666667     29   0.0   -0.166667

tol  limit
0.001  25

incs, (qinc(i),i=1,incs)
10
200.0 100.0 50.0 50.0 50.0 30.0 20.0 10.0 5.0 4.0

```

**Figure 6.34** Data for Program 6.8 example

For the special case of non-pressure-dependent (cylindrical) failure criteria such as von Mises, the general CPPM algorithm described by Program 6.6 reduces to the well-known radial return method (e.g., Krieg and Krieg, 1977; Belytchko *et al.*, 2000) described here as Program 6.8. When stresses stray outside the failure surface, such as point *B* in Figure 6.27, the method adjusts the deviatoric stress components analytically (6.64) such that the stresses return exactly to the failure surface. With the added refinement of a two-step ‘line search’ correction (e.g., Crisfield, 1997), the resulting algorithm is efficient and robust.

The example problem is the same as the one considered previously in Programs 6.1, 6.6 and 6.7. The data given in Figure 6.34 is exactly the same as in Figure 6.9, except without the *nf* data, which has once more been dealt with by the boundary condition subroutine *bc\_rect*. The iteration ceiling *limit* can also be set at a much lower value than was needed in Figure 6.9. The results using the radial return algorithm, shown in Figure 6.35, are seen to be identical to those obtained by direct solution with CPPM in Figure 6.30, but using less computer time.

## 6.11 The Geotechnical Processes of Embanking and Excavation

### 6.11.1 Embanking

One of the main features of analyses of geotechnical problems is the need to model construction processes. Gravity is one of the main agencies causing deformations and it is common to employ ‘gravity turn-on’ as the loading mechanism. In embankments, for example, the final geometry of a weightless slope can be modelled by a finite element

```

There are      192 equations and the skyline storage is    4322

step    load      disp      iters
 1  0.2000E+03 -0.6593E-02    2
 2  0.3000E+03 -0.1161E-01    3
 3  0.3500E+03 -0.1635E-01    3
 4  0.4000E+03 -0.2319E-01    3
 5  0.4500E+03 -0.3326E-01    3
 6  0.4800E+03 -0.4259E-01    3
 7  0.5000E+03 -0.5140E-01    3
 8  0.5100E+03 -0.5730E-01    3
 9  0.5150E+03 -0.6265E-01    3
10  0.5190E+03 -0.1510E+00    6
Time taken =          0.0624

```

**Figure 6.35** Results from Program 6.8 example

mesh, which is then subjected to gravity loading, often in a single increment. To obtain the factor of safety, the soil's strength parameters can be reduced sequentially to failure (see, e.g., Program 6.4).

Although it has been known for a long time (Smith and Hobbs, 1974) that this method can capture some of the realities of a construction process which takes place piece by piece (in layers, for example), it is more realistic to be able to build up a mesh sequentially, modelling the influence of gravity at each stage.

### 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

```

PROGRAM p69
!-----
! 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.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::enxe,eny,e,f,fnxe,fnye,i,iel,ii,incs,iters,itype,iy,k,lifts,limit, &
  lnn,ndim=2,ndof=16,nels,neq,newele,nip=4,nlen,nn,nod=8,nodof=2,nr,       &
  nst=4,oldele,oldnn
REAL(iwp)::c,c_e,c_f,det,dq1,dq2,dq3,dsbar,dt,d4=4.0_iwp,d180=180.0_iwp, &
  e,e_e,e_f,f,gamma,gama_e,gama_f,k0,lode_theta,one=1.0_iwp,phi,phi_e,      &
  phi_f,pi,psi,psi_e,psi_f,sigm,snph,tol,two=2.0_iwp,v,v_e,v_f,            &
  zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:),g_g(:, :, ),g_num(:, :, ),kdiag(:,lnf(:, :, ), &
  nf(:, :, ),num(:, ))
REAL(iwp),ALLOCATABLE::bdylds(:,bee(:, :, ),bload(:, ),coord(:, :, ),dee(:, :, ), &
  der(:, :, ),deriv(:, :, ),devp(:, ),edepth(:, ),eld(:, ),eload(:, ),eps(:, ),erate(:, ), &
  evp(:, ),evpt(:, :, :, ),ewidth(:, ),fdepth(:, ),flow(:, :, ),fun(:, ),fwidth(:, ), &
  gc(:, ),gravlo(:, ),g_coord(:, :, ),jac(:, :, ),km(:, :, ),kv(:, ),loads(:, ),m1(:, :, ), &
  m2(:, :, ),m3(:, :, ),oldis(:, ),points(:, :, ),sigma(:, ),stress(:, :, :, ), &
  totd(:, ),weights(:, )

```

```

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*) fnxe,fnye,nn,incs,limit,tol,lifts,enxe,enye,itype,k0,e_f,v_f, &
c_f,phi_f,psi_f,gama_f,e_e,v_e,c_e,phi_e,psi_e,gama_e
!-----calculate the total number of elements-----
k=0; DO i=1,enye-1; k=i+k; END DO; nels=fnxe*fnye+(enxe*enye-k)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
edepth(enye+1),num(nod),dee(nst,nst),evpt(nst,nip,nels),ewidth(enxe+1), &
coord(nod,ndim),fun(nod),etype(nels),g_g(ndof,nels),jac(ndim,ndim), &
der(ndim,nod),deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof), &
km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof),eload(ndof), &
erate(nst),evp(nst),devp(nst),g(ndof),ml(nst,nst),m2(nst,nst), &
m3(nst,nst),flow(nst,nst),stress(nst),fwidth(fnxe+1),fdepth(fnye+1), &
gc(ndim),tensor(nst,nip,nels))
READ(10,*) fwidth,fdepth,ewidth,edepth
nf=1; READ(10,*)nr,(k,nf(:,k),i=1, nr); CALL formnf(nf); neq=MAXVAL(nf)
WRITE(11,'(A,I5)')" The final number of elements is:",nels
WRITE(11,'(A,I5)')" The final number of freedoms is:",neq
!-----set the element type-----
etype(1:fnxe*fnye)=1; etype(fnxe*fnye+1:nels)=2
!-----set up the global node numbers and element nodal coordinates--
CALL fmglem(fnxe,fnye,enxe,g_num,lifts)
CALL fmcoem(g_num,g_coord,fwidth,fdepth,ewidth,edepth, &
enxe, lifts,fnxe,fnye,itype)
ALLOCATE(totd(0:neq)); tensor=zeros; totd=zeros; pi=ACOS(-one)
!-----loop the elements to find the global g-----
elements_1: DO iel=1,nels
    num=g_num(:,iel); CALL num_to_g(num,nf,g); g_g(:,iel)=g
END DO elements_1; CALL sample(element,points,weights)
!-----construct another lift-----
lift_number: DO ii=1, lifts
    WRITE(11,'(/A,I5)')" Lift number",ii
!-----calculate how many elements there are-----
IF(ii<=lifts)THEN
    IF(ii==1)THEN; newele=fnxe*fnye; oldele=newele; ELSE
        newele=enxe-(ii-2); oldele=oldele+newele
    END IF
!-----calculate how many nodes there are-----
IF(ii==1)THEN; lnn=(fnxe*2+1)*(fnye+1)+(fnxe+1)*fnye; oldnn=lnn
END IF
IF(ii>1)THEN; lnn=oldnn+(enxe-(ii-2))*2+1+(enxe-(ii-2)+1); oldnn=lnn
END IF; ALLOCATE(lnf(nodof,lnn)); lnf=nf(:,1:lnn)
!-----recalculate the number of freedoms neq-----
neq=MAXVAL(lnf); ALLOCATE(kdiag(neq)); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_2: DO iel=1,oldele; g=g_g(:,iel); CALL fkdiag(kdiag,g)
END DO elements_2
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
WRITE(11,'(3(A,I5))')" There are",neq," freedoms"
WRITE(11,'(3(A,I5))')" There are",oldele," elements after",
    newele," were added"
END IF
ALLOCATE(kv(kdiag(neq)),loads(0:neq),bdylds(0:neq),oldis(0:neq), &
gravlo(0:neq)); gravlo=zeros; loads=zeros; kv=zeros
!-----element stiffness integration and assembly-----
elements_3: DO iel=1,oldele
    IF(etype(iel)==1)THEN; gamma=gama_f; e=e_f; v=v_f

```

```

ELSE; gamma=gama_e; e=e_e; v=v_e
END IF
IF(iel<=(oldele-newele))gamma=zero; num=g_num(:,iel)
coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
CALL deemat(dee,e,v); eld=zero
gauss_pts_1: DO i=1,nip
    CALL shape_fun(fun,points,i); gc=MATMUL(fun,coord)
!-----initial stress in foundation-----
    IF(ii==1)THEN; tensor(2,i,iel)=-one*(fdepth(fnye+1)-gc(2))*gamma
        tensor(1,i,iel)=k0*tensor(2,i,iel)
        tensor(4,i,iel)=tensor(1,i,iel); tensor(3,i,iel)=zero
    END IF
    CALL shape_der(der,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    DO k=2,ndof,2; eld(k)=eld(k)+fun(k/2)*det*weights(i); END DO
    END DO gauss_pts_1
    CALL fsparv(kv,km,g,kdiag)
    IF(ii<=lifts)gravlo(g)=gravlo(g)-eld*gamma; gravlo(0)=zero
    END DO elements_3
!-----factorise equations--and-factor gravlo by incs---
    CALL sparin(kv,kdiag); gravlo=gravlo/incs
!-----apply gravity loads incrementally-----
load_incs: DO iy=1,incs
    iters=0; oldis=zero; bdylds=zero; evpt(:,:,1:oldele)=zero
!-----iteration loop-----
    its: DO
        iters=iters+1
        WRITE(*,'(A,I3,A,I3,A,I4)')" lift",ii," increment",iy,
        " iteration",iters
        loads=zero; loads=gravlo+bdylds; CALL spabac(kv,loads,kdiag)
!-----check convergence-----
        CALL checon(loads,oldis,tol,converged)
        IF(iters==1)converged=.FALSE.
        IF(converged.OR.iters==limit)bdylds=zero
!-----go round the Gauss Points-----
elements_4: DO iel=1,oldele
    IF(etype(iel)==1)THEN; phi=phi_f; c=c_f; e=e_f; v=v_f; psi=psi_f
    ELSE; phi=phi_e; c=c_e; e=e_e; v=v_e; psi=psi_e
    END IF; snph=SIN(phi*pi/d180)
    dt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2))
    CALL deemat(dee,e,v); bload=zero; num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
    gauss_pts_2: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        eps=MATMUL(bee,eld); eps=eps-evpt(:,:,iel)
        sigma=MATMUL(dee,eps)
        IF(ii==1)THEN; stress=tensor(:,:,iel)
        ELSE; stress=tensor(:,:,iel)+sigma
        END IF; CALL invar(stress,sigm,dsbar,lode_theta)
!-----check whether yield is violated-----
        CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
        IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
            IF(f>=zero)THEN
                CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)

```

```

        CALL formm(stress,m1,m2,m3); flow=f* (m1*dq1+m2*dq2+m3*dq3)
        erate=MATMUL(flow,stress); evp=erate*dt
        evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee, evp)
    END IF
END IF
IF(f>=zero)THEN; eload=MATMUL(devp,bee)
    bload=bload+eload*det*weights(i)
END IF
!-----if appropriate update the Gauss point stresses-----
IF(converged.OR.iters==limit)THEN
    IF(ii/=1)tensor(:,i,iel)=stress
END IF
END DO gauss_pts_2
!-----compute the total bodyloads vector-----
bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
END DO elements_4; IF(converged.OR.iters==limit)EXIT
END DO its; IF(ii/=1)totd(:neq)=totd(:neq)+loads(:neq)
WRITE(11,'(2(A,I5),A)')" Increment",iy," took ",iters,
    &
    " iterations to converge"
IF(iy==incs.OR.iters==limit)WRITE(11,'(A,E12.4)')
    &
    " Max displacement is",MAXVAL(ABS(loads))
IF(iters==limit)THEN
    CALL dismsh(loads,lnf,0.05_iwp,g_coord,g_num,argv,nlen,13)
    CALL vecmsh(loads,lnf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
    STOP
END IF
END DO load_incs; DEALLOCATE(lnf,kdiag,kv,loads,bdylds,oldis,gravlo)
END DO lift_number
STOP
END PROGRAM p69

```

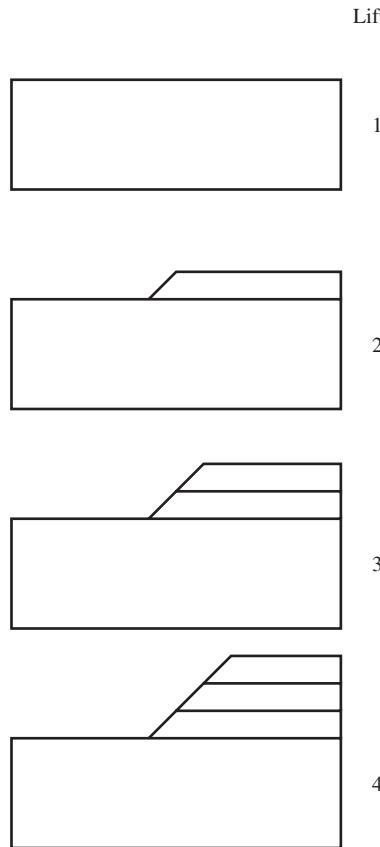
Program 6.9 analyses the stability of a slope, built up in layers on a foundation. Figure 6.36 shows a typical staged construction of an embankment (plane strain) on a rectangular foundation block of soil.

The embankment is assumed to be raised in a series of ‘lifts’, the first of which merely stresses the foundation block gravitationally under ‘at rest’ conditions. The ‘soil’ is assumed to be an elastic–plastic Mohr–Coulomb material and a viscoplastic strain algorithm is used, so the program can be considered to be a development of Program 6.4. Figure 6.37 shows the final mesh in more detail, together with the input data for the problem.

The basic mesh information is read in the data with fnxe, fnye, fwidth and fdepth for the foundation, and enxe, enye, ewidth and edepth for the embankment. Subroutines fmglem and fmcoem set up the global node numbers and element nodal coordinates g\_num and g\_coord, respectively, in this case customised for a 2:1 slope inclined at  $26.57^\circ$  to the horizontal. Alternatives, controlled by itype as shown in the figure, allow quadrilaterals to be degenerated into triangles (on the face of the embankment slope) by two different methods.

The problem consists of two materials, the foundation, with properties given by e\_f, v\_f, c\_f, phi\_f, psi\_f and gama\_f and the embankment, with properties given by e\_e, v\_e, c\_e, phi\_e, psi\_e and gama\_e.

Because the mesh is updated at every ‘lift’, there is a need for a ‘local’ node freedom array lnf which is found from the final nf at every stage. lnf is ALLOCATED and DEALLOCATED at each lift. Then, for each lift, the geometry and connectivity can be



**Figure 6.36** Staged construction of an embankment

calculated and hence the number of equations `neq` and stiffness matrix diagonal locator `kdiag`, operating at that stage of the construction process. The stiffness matrix and load vector sizes can then be set by an `ALLOCATE` statement and the viscoplastic algorithm initiated. The program has the option of applying the gravity loads associated with each lift in `incs` increments (three in this case).

The results are shown in Figure 6.38 and plotted in Figure 6.39, showing the progress of toe displacement as the embankment is raised. For the case shown, in which the foundation and embankment are both undrained clays, with  $c_u = 14 \text{ kN/m}^2$ , the embankment is seen to fail when its height reaches approximately 4 m. This is very similar to what would be predicted by Taylor's (1937) charts, which give, for a 2:1 slope with a depth ratio of  $D = 3.5$ , a stability number equal to 0.18, implying a factor of safety very close to unity. The displacement vectors corresponding to the un converged solution when the embankment height reached 4 m are shown in Figure 6.40.

### 6.11.2 Excavation

The second important geotechnical construction process involving change of geometry occurs when material is removed from the ground, either in open excavations ('cuts') or

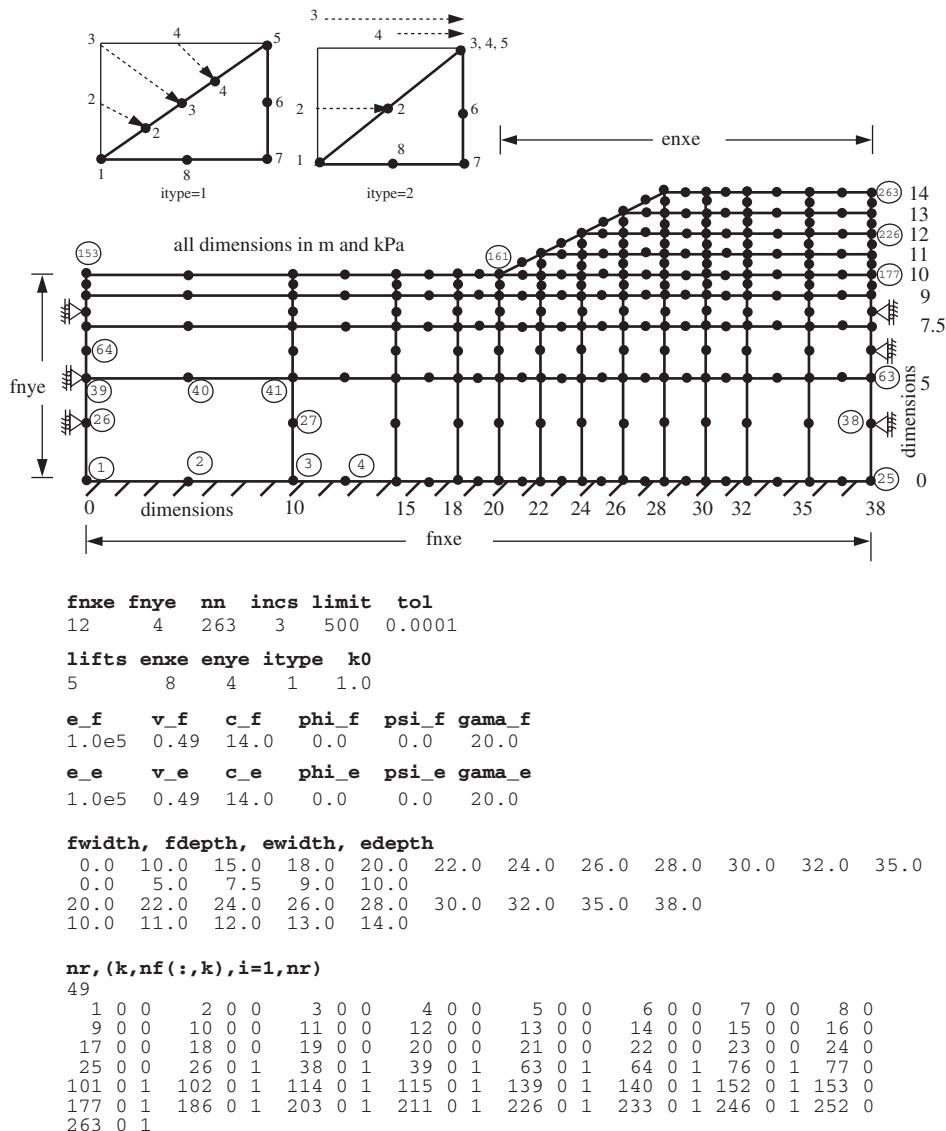


Figure 6.37 Mesh and data for Program 6.9 example

in enclosed tunnels. The ground is stressed prior to removal of part of it and this starting stress state may be difficult to infer from the known history.

The aim in an analysis is that, when a portion of material is excavated, and forces are applied along the excavated surface, the remaining material should experience the correct stress relief so that the new ‘free surface’ is indeed stress-free.

Suppose body  $A$  is to be removed from body  $B$  as shown in Figure 6.41. The stresses to begin with are  $\{\sigma_{A0}\}$  and  $\{\sigma_{B0}\}$ , respectively.

```

The final number of elements is: 74
The final number of freedoms is: 452

Lift number 1
There are 288 freedoms
There are 48 elements after 48 were added
Increment 1 took 2 iterations to converge
Increment 2 took 2 iterations to converge
Increment 3 took 2 iterations to converge
Max displacement is 0.1948E-03

Lift number 2
There are 338 freedoms
There are 56 elements after 8 were added
Increment 1 took 2 iterations to converge
Increment 2 took 2 iterations to converge
Increment 3 took 2 iterations to converge
Max displacement is 0.2624E-03

Lift number 3
There are 382 freedoms
There are 63 elements after 7 were added
Increment 1 took 2 iterations to converge
Increment 2 took 2 iterations to converge
Increment 3 took 2 iterations to converge
Max displacement is 0.2806E-03

Lift number 4
There are 420 freedoms
There are 69 elements after 6 were added
Increment 1 took 2 iterations to converge
Increment 2 took 5 iterations to converge
Increment 3 took 7 iterations to converge
Max displacement is 0.3311E-03

Lift number 5
There are 452 freedoms
There are 74 elements after 5 were added
Increment 1 took 15 iterations to converge
Increment 2 took 45 iterations to converge
Increment 3 took 500 iterations to converge
Max displacement is 0.1138E+00

```

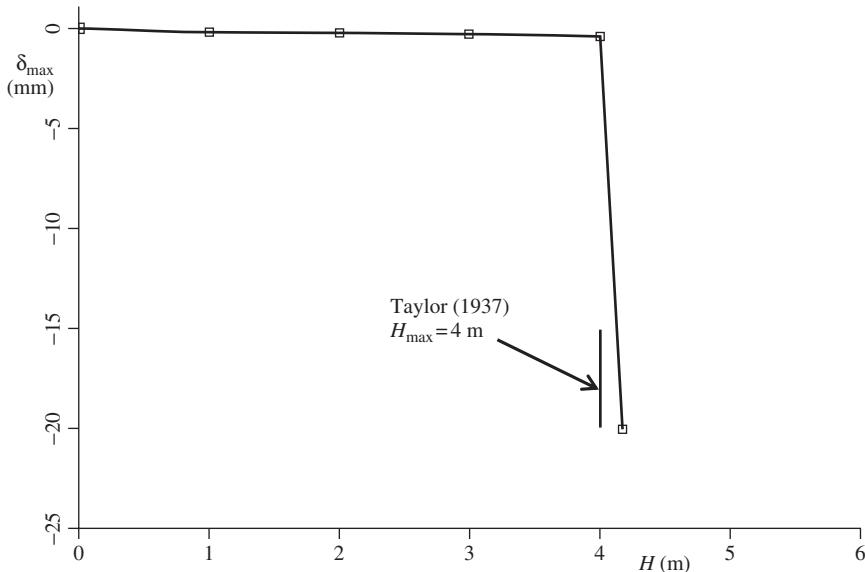
**Figure 6.38** Results from Program 6.9 example

Any external loads are taken into consideration in forming these stresses prior to the removal of A. Since both bodies are in equilibrium, forces  $\{\mathbf{F}_{AB}\}$  must be applied to body B due to body A to maintain  $\{\sigma_{B0}\}$  and, similarly,  $\{\mathbf{F}_{BA}\}$  must act on body A. Forces  $\{\mathbf{F}_{AB}\}$  and  $\{\mathbf{F}_{BA}\}$  are equal in magnitude and opposite in sign. In general, therefore, the excavation forces acting on a boundary depend on the stress state in the excavated material and on the self-weight of that material. It can be shown that

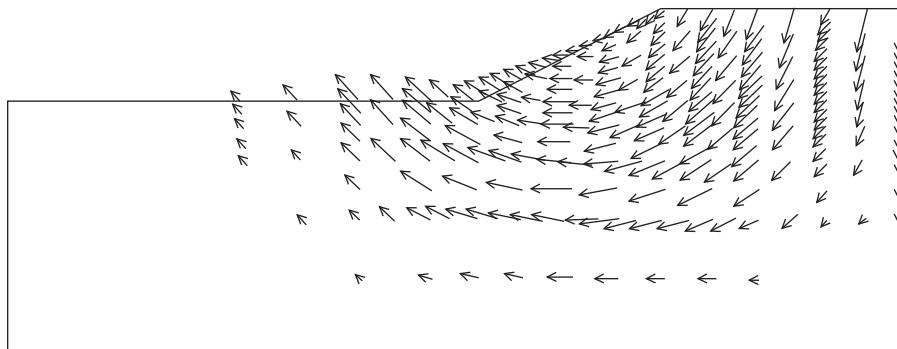
$$\{\mathbf{F}_{BA}\} = \int_{V_A} [\mathbf{B}]^T \{\boldsymbol{\sigma}_{A0}\} dV_A + \gamma \int_{V_A} [\mathbf{N}]^T dV_A \quad (6.75)$$

where  $[\mathbf{B}]$  is the strain-displacement matrix,  $V_A$  the excavated volume,  $[\mathbf{N}]$  the element shape functions and  $\gamma$  the soil unit weight.

Single-stage and multi-stage excavations give the same results, and in a one-dimensional situation a stress-free excavated surface results. Figure 6.42 shows a column of five



**Figure 6.39** Embankment height vs. maximum nodal displacement from Program 6.9 example

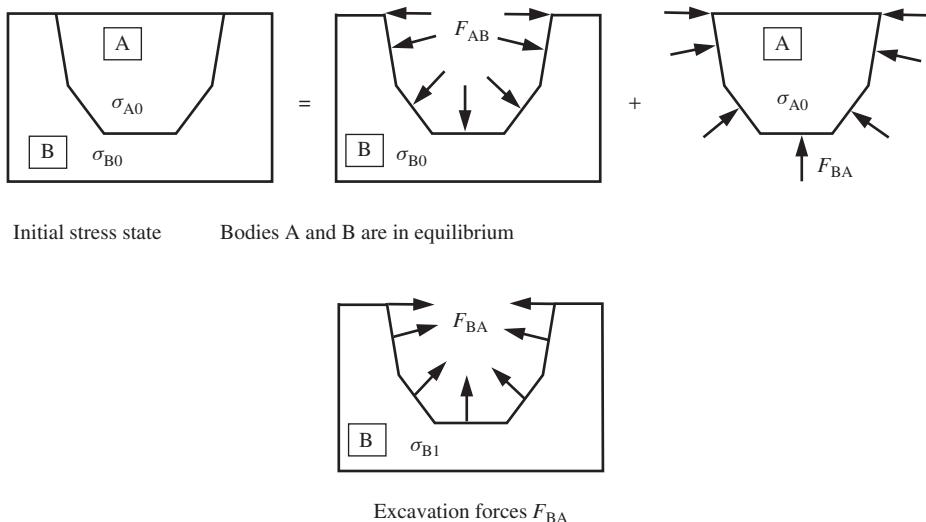


**Figure 6.40** Displacement vectors at failure when embankment height reaches 4 m from Program 6.9 example

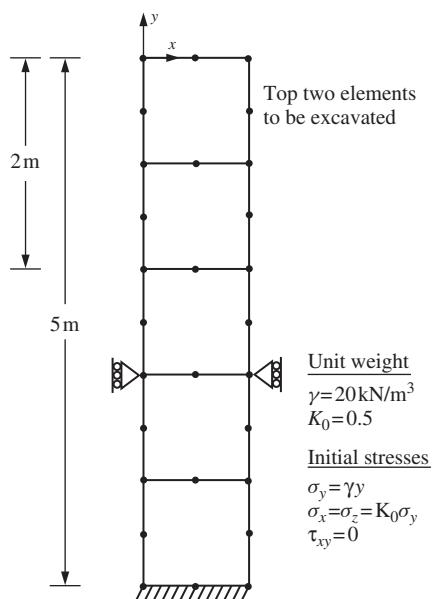
8-node elements at rest under self-weight stresses from which the top two elements are to be excavated. Figure 6.43 shows the excavation force terms computed using  $2 \times 2$  Gauss points from (6.75), and the resulting forces that need to be applied to the free surface.

When the situation is two- or three-dimensional, equation (6.75) applies again, but in the corners of excavations, a rather complex stress concentration exists. This means that the finite element results will be mesh-dependent (Smith and Ho, 1992).

A program illustrating the analysis of excavation processes is listed as Program 6.10. As with the previous program, it can be derived from Program 6.4.



**Figure 6.41** Formulation of excavation forces



**Figure 6.42** Column of elements before excavation

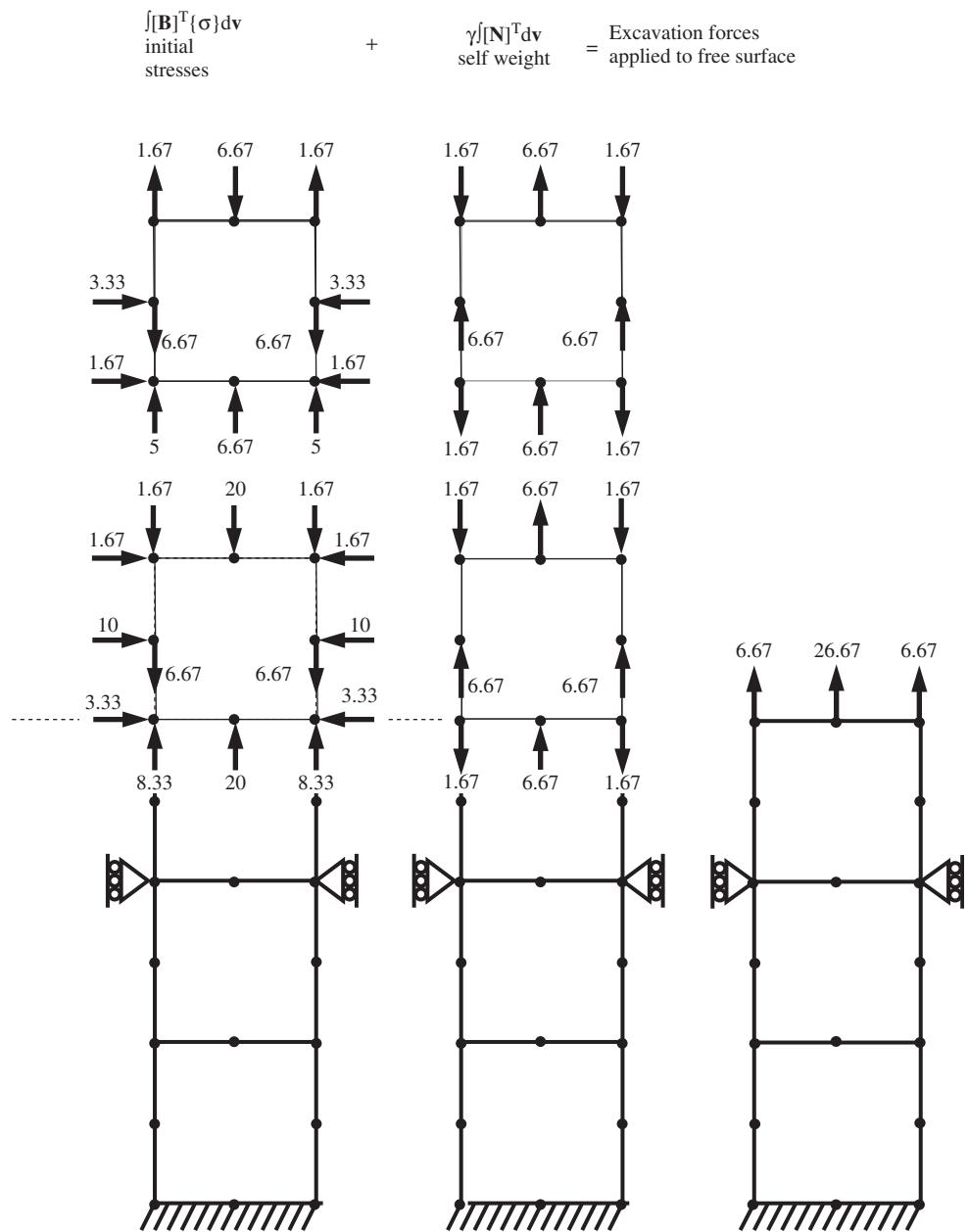


Figure 6.43 Development of excavation force terms

## Program 6.10 Plane-strain construction of an elastic–plastic (Mohr–Coulomb) excavation in layers using 8-node quadrilaterals. Viscoplastic strain method

```

PROGRAM p610
!-----
! Program 6.10 Plane strain construction of an elastic-plastic
!           (Mohr-Coulomb) excavation in layers using 8-node
!           quadrilaterals. Viscoplastic strain method.
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,ii,incs,iters,iy,k,layers,limit,ndim=2,ndof=16,nels,neq,  &
  nip=4,nlen,nn,nod=8,nodof=2,noexe,nouts,nprops=7,np_types,nr,nst=4,  &
  ntote
REAL(iwp)::c,ddt,det,dq1,dq2,dq3,dsbar,dt,d4=4.0_iwp,d180=180.0_iwp,e,f,  &
  gamma,lode_theta,one=1.0_iwp,phi,pi,psi,sigm,snph,start_dt=1.e15_iwp,  &
  tol,two=2.0_iwp,v,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),exele(:),g(:,),g_num(:,,:),kdiag(:,),lnf(:,,:),  &
  nf(:,,:),no(:,),num(:,),solid(:,),totex(:,)
REAL(iwp),ALLOCATABLE::bdylds(:,),bee(:, :,),bload(:, ),coord(:, :,),dee(:, :,),  &
  der(:, :,),deriv(:, :,),devp(:, ),eld(:, ),eload(:, ),eps(:, ),erate(:, ),evp(:, ),  &
  evpt(:, :, :),exc_loads(:, ),flow(:, :,),fun(:, ),gc(:, ),g_coord(:, :,),jac(:, :,),  &
  km(:, :,),kv(:, ),loads(:, ),m1(:, :,),m2(:, :,),m3(:, :,),oldis(:, ),points(:, :,),  &
  prop(:, :,),stress(:, ),tensor(:, :, :,),tot_d(:, :,),weights(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,nn,np_types; ALLOCATE(prop(nprops,np_types),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),  &
  num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod),  &
  solid(nels),jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),  &
  g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof),eps(nst),  &
  totex(nels),bload(ndof),eload(ndof),erate(nst),evp(nst),devp(nst),  &
  g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),stress(nst),  &
  tot_d(nodof,nn),gc(ndim),tensor(nst,nip,nels),lnf(nodof,nn))
!-----read geometry and connectivity-----
READ(10,*)g_coord,g_num; nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr)
!-----lnf is the current nf at each stage of excavation-----
lnf=nf; CALL formnf(lnf); neq=MAXVAL(lnf)
WRITE(11,'(A,I5)')" The initial number of elements is:",nels
WRITE(11,'(A,I5)')" The initial number of freedoms is:",neq
!-----set up the global node numbers and global nodal coordinates-----
!-----loop the elements to set starting stresses-----
CALL sample(element,points,weights)
elements_0: DO iel=1,nels
  num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
  gamma=prop(4,etype(iel))
  gauss_pts_0: DO i=1,nip
    CALL shape_fun(fun,points,i); gc=MATMUL(fun,coord)
    tensor(2,i,iel)=gc(2)*gamma
    tensor(1,i,iel)=prop(7,etype(iel))*tensor(2,i,iel)
    tensor(4,i,iel)=tensor(1,i,iel); tensor(3,i,iel)=zero
  END DO gauss_pts_0
END DO

```

```

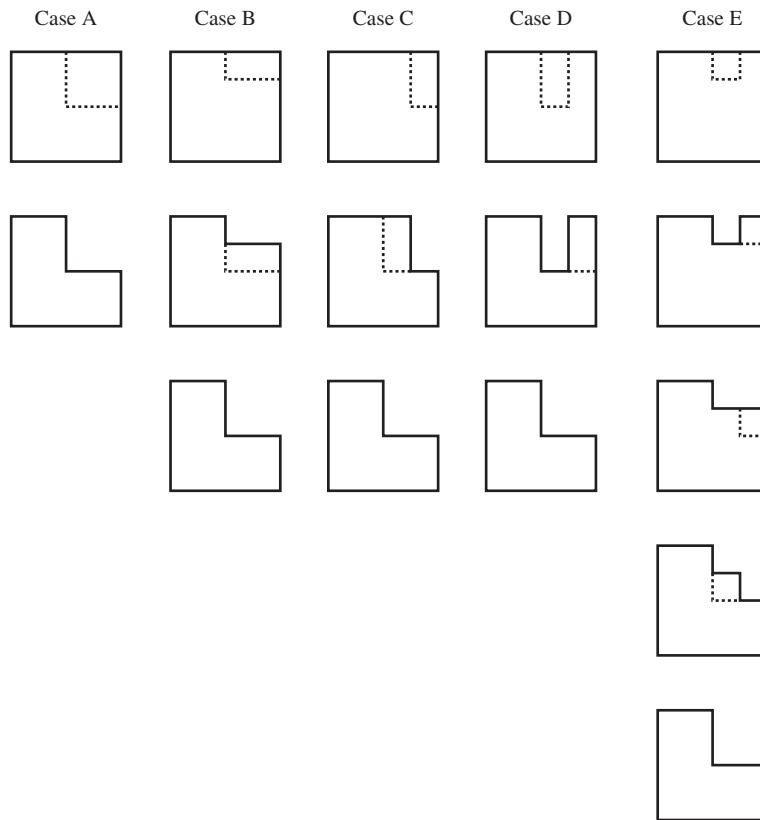
END DO elements_0; tot_d=zero; ntote=0; solid=1; totex=0; pi=ACOS(-one)
!-----excavate a layer-----
READ(10,*)nouts; ALLOCATE(no(nouts)); READ(10,*)no,tol,limit,incs,layers
layer_number: DO ii=1,layers
    WRITE(11,'(/A,I5)')" Excavation number",ii
!-----read elements to be removed-----
    READ(10,*)noexe; ALLOCATE(exele(noexe)); READ(10,*)exele; solid(exele)=0
    CALL exc_nods(noexe,exele,g_num,totex,ntote,nf); lnf=nf
    CALL formnf(lnf); neq=MAXVAL(lnf)
    ALLOCATE(kdiag(neq),exc_loads(0:neq),bdylds(0:neq),oldis(0:neq),
              &
              loads(0:neq)); WRITE(11,'(3(A,I5))')" There are",neq," freedoms"
    WRITE(11,'(3(A,I5))')" There are",nels-ntote," elements after",
              &
              noexe," were removed"; kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    num=g_num(:,iel); CALL num_to_g(num,lnf,g); CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq))); exc_loads=zero
!-----calculate excavation load -----
elements_2: DO iel=1,noexe
    k=exele(iel); gamma=prop(4,etype(k)); bload=zero; eld=zero
    num=g_num(:,k); CALL num_to_g(num,lnf,g)
    coord=TRANSPOSE(g_coord(:,num))
    gauss_pts_2: DO i=1,nip
        CALL shape_fun(fun,points,i); stress=tensor(:,i,k)
        CALL bee8(bee,coord,points(i,1),points(i,2),det)
        eload=MATMUL(stress,bee); bload=bload+eload*det*weights(i)
        eld(nodof:nodof)=eld(nodof:nodof)+fun(:)*det*weights(i)
    END DO gauss_pts_2; exc_loads(g)=exc_loads(g)+eld*gamma+bload
END DO elements_2; exc_loads(0)=zero
!-----element stiffness integration and assembly-----
kv=zero; dt=start_dt
elements_3: DO iel=1,nels
    IF(solid(iel)==0)THEN; e=zero; ELSE
        phi=prop(1,etype(iel)); e=prop(5,etype(iel)); v=prop(6,etype(iel))
        snph=SIN(phi*pi/d180)
        ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph*snph))
        IF(ddt<dt)dt=ddt
    END IF
    km=zero; eld=zero; CALL deemat(dee,e,v); num=g_num(:,iel)
    CALL num_to_g(num,lnf,g); coord=TRANSPOSE(g_coord(:,num))
    gauss_pts_3: DO i=1,nip
        CALL bee8(bee,coord,points(i,1),points(i,2),det)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_3; CALL fsparv(kv,km,g,kdiag)
END DO elements_3
!-----factorise sand factor excavation load by incs-----
CALL sparin(kv,kdiag); exc_loads=exc_loads/incs
!-----apply excavation loads incrementally-----
load_incs: DO iy=1,incs
    iters=0; oldis=zero; bdylds=zero; evpt=zero
!-----iteration loop -----
    its: DO
        iters=iters+1
        WRITE(*,'(A,I3,A,I3,A,I4)')" excavation",ii," increment",iy,
              &
              " iteration",iters
        loads=exc_loads+bdylds; CALL spabac(kv,loads,kdiag)
!-----check convergence-----

```

```

CALL checon.loads,oldis,tol,converged)
IF(iters==1) converged=.FALSE.
IF(converged.OR.iters==limit)THEN; bdylds=zero
    DO k=1,nm; DO i=1,nodof
        IF(lnf(i,k)/=0)tot_d(i,k)=tot_d(i,k)+loads(lnf(i,k))
    END DO; END DO
END IF
!-----go round the Gauss Points-----
elements_4: DO iel=1,nel
    phi=prop(1,etype(iel)); c=prop(2,etype(iel))
    psi=prop(3,etype(iel)); e=prop(5,etype(iel))
    v=prop(6,etype(iel)); IF(solid(iel)==0)e=zero; bload=zero
    CALL deemat(dee,e,v); num=g_num(:,iel); CALL num_to_g(num,lnf,g)
    coord=TRANSPOSE(g_coord(:,num)); eld=loads(g)
    gauss_pts_4: DO i=1,nip
        CALL bee8(bee,coord,points(i,1),points(i,2),det)
        eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
        stress=tensor(:,i,iel)+MATMUL(dee,eps)
    !-----air element stresses are zero-----
        IF(solid(iel)==0)stress=zero
        CALL invar(stress,sigm,dsbar,lode_theta)
    !-----check whether yield is violated-----
        CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
        IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
            IF(f>=zero)THEN; CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
            CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
            erate=MATMUL(flow,stress); evp=erate*dt
            evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
        END IF
        END IF
        IF(f>=zero.OR.(converged.OR.iters==limit))THEN
            eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
        END IF
    !-----if appropriate update the Gauss point stresses---
        IF(converged.OR.iters==limit)tensor(:,i,iel)=stress
    END DO gauss_pts_4
!-----compute the total bodyloads vector -----
    bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
    END DO elements_4; IF(converged.OR.iters==limit)EXIT
    END DO its
    WRITE(11,'(A,I3,A,I5,A)')" Increment",iy," took",iters,
    " iterations to converge" &
    IF(iy==incs.OR.iters==limit)THEN
        WRITE(11,'(A)')" Node x-disp y-disp"
        DO i=1,nouts; WRITE(11,'(I5,2E12.4)')no(i),tot_d(:,no(i)); END DO
        EXIT
    END IF
    END DO load_incs; IF(ii==layers.OR.iters==limit)EXIT
    DEALLOCATE(kdiag,kv,exc_loads,bdylds,oldis,loads,exele)
END DO layer_number
loads(lnf(1,:))=tot_d(1,:); loads(lnf(2,:))=tot_d(2,:)
g_num(:,totex(:ntote))=0; CALL mesh(g_coord,g_num,argv,nlen,12)
CALL dismsh(loads,lnf,0.1_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,lnf,0.1_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p610

```

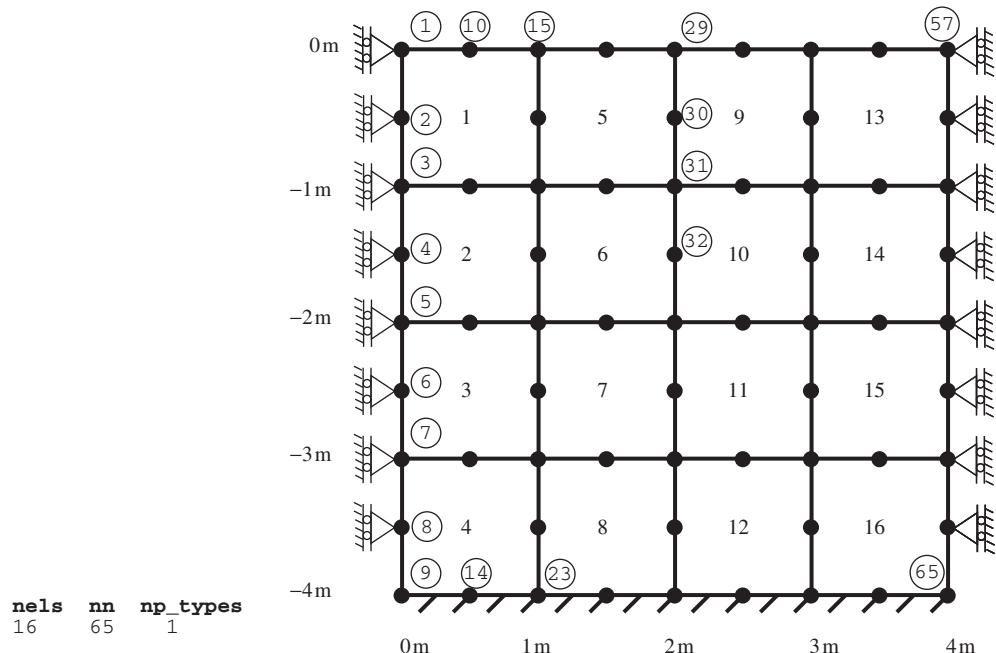


**Figure 6.44** Five ways of excavating a vertical cut

Figure 6.44 shows a square block of ‘soil’ from which a vertical cut is to be excavated. The figure indicates five possible sequences that would lead to the same final excavation geometry. Figure 6.45 shows the mesh and data corresponding to excavation case *B* in a soil with undrained shear strength  $c_u = 9 \text{ kN/m}^2$ .

This program allows excavations over general shapes of meshes, so it is the user’s responsibility to provide the global coordinates `g_coord` and element numbering `g_num` of the original unexcavated configuration as data. Figure 6.45 shows a truncated data set in the interests of a compact presentation. The same concept of a ‘local’ node freedom array `lnf` as in the previous program is used again. In a preliminary loop, labelled `elements_0`, the ground is stressed by its own weight. The soil model allows for seven properties read as usual into the array `prop`. Note that due to the simplicity of the constitutive model, a very limited range of  $K_0$  values could be achieved automatically, so  $K_0$  is input as data as the seventh property read into the array `prop`.

Following the coordinates, element numbering and boundary condition data, the data requires the number of nodes at which output will be required `nouts`, followed by the output node numbers `no`, the plastic tolerance `tol`, the iteration ceiling limit, the number of load increments per excavation `incs` and the number of excavations `layers`.



**etype**(not needed)

**g\_coord**

0.00	0.00	0.00	-0.50	0.00	-1.00	0.00	-1.50	0.00	-2.00
0.00	-2.50	0.00	-3.00	0.00	-3.50	0.00	-4.00	0.50	0.00

g\_coord data for nodes 1155 have been omitted here  
3.50 -4.00 4.00 0.00 4.00 -0.50 4.00 -1.00 4.00 -1.50  
4.00 -2.00 4.00 -2.50 4.00 -3.00 4.00 -3.50 4.00 -4.00

**g\_num**

3	2	1	10	15	16	17	11
5	4	3	11	17	18	19	12

g\_num data for elements 314 have been omitted here  
49 48 47 54 61 62 63 55  
51 50 49 55 63 64 65 56

**nr,(k,nf(:,k),i=1,nr)**

25	1 0 1	2 0 1	3 0 1	4 0 1	5 0 1	6 0 1	7 0 1	8 0
9 0 0	14 0 0	23 0 0	28 0 0	37 0 0	42 0 0	51 0 0	56 0	
57 0 1	58 0 1	59 0 1	60 0 1	61 0 1	62 0 1	63 0 1	64 0	
65 0 0								

**nouts,(no(i),i=1,nouts)**

4  
29 30 31 32

**tol limit incs layers**  
0.0001 250 5 2

**noexe,(exe(i),i=1,noexe) (for excavation 1)**  
2  
9 13

**noexe,(exe(i),i=1,noexe) (for excavation 2)**  
2  
10 14

**Figure 6.45** Mesh and data for Program 6.10 example

```

The initial number of elements is: 16
The initial number of freedoms is: 96

Excavation number      1
There are    86 freedoms
There are    14 elements after    2 were removed
Increment  1 took      2 iterations to converge
Increment  2 took      2 iterations to converge
Increment  3 took      2 iterations to converge
Increment  4 took      2 iterations to converge
Increment  5 took      2 iterations to converge
Node   x-disp       y-disp
 29  0.7635E-05 -0.5876E-04
 30  0.9240E-04 -0.3784E-04
 31  0.8605E-04 -0.1057E-04
 32  0.1102E-03 -0.1652E-05

Excavation number      2
There are    76 freedoms
There are    12 elements after    2 were removed
Increment  1 took      2 iterations to converge
Increment  2 took      2 iterations to converge
Increment  3 took      5 iterations to converge
Increment  4 took     22 iterations to converge
Increment  5 took    250 iterations to converge
Node   x-disp       y-disp
 29 -0.3073E-03 -0.1641E-02
 30  0.9790E-03 -0.1531E-02
 31  0.2371E-02 -0.1282E-02
 32  0.4044E-02 -0.7897E-03

```

**Figure 6.46** Results from Program 6.10 example

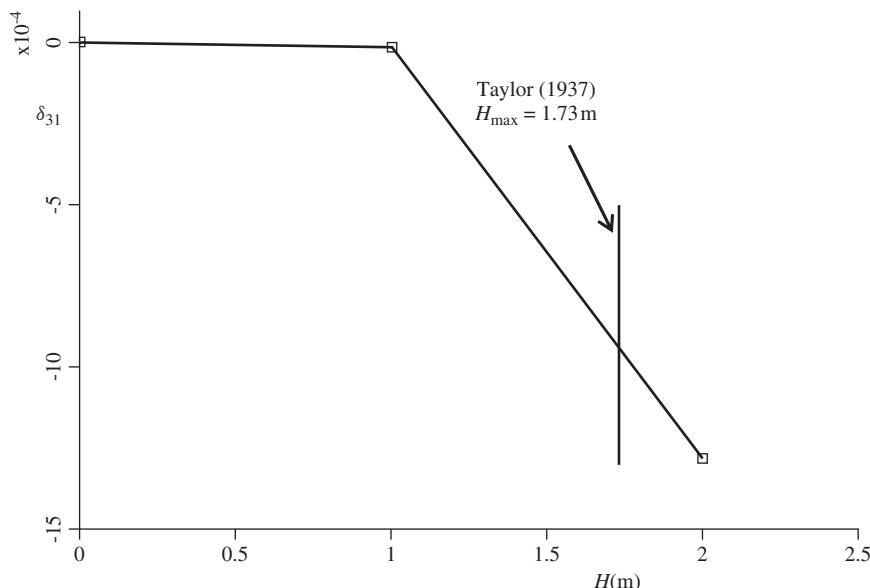
The final block of data gives the excavation sequence. For each of the excavation steps, the number of elements to be removed `noexe` and the element numbers `exele` must be read. The subroutine `exc_nods` computes the node numbers removed at each excavation step. Excavated ‘air’ elements are given a stiffness of zero ( $E = 0$ ) and the excavated nodes are automatically fully restrained and hence removed from the assembly process.

As in the previous program, for each ‘layer’ the geometry is modified and a new stiffness matrix and load vectors formed. Arrays that change their size from one excavation to the next therefore involve `ALLOCATE` and `DEALLOCATE` statements.

For the case considered, the vertical excavation (case *B*) is to occur in two steps `layers=2`, each split into five equal increments `incs=5`, leading to a cut of depth 2 m. As can be seen from the data, the first excavation removes elements 9 and 13, and the second excavation removes elements 10 and 14. The output (for case *B*) shown as Figure 6.46 and plotted in Figure 6.47, indicates that the displacement at node 31 (on the excavation face) increased very significantly after the second excavation. For a vertical cut consisting of undrained clay with a strength of 9 kN/m<sup>2</sup>, Taylor (1937) predicts a critical height of approximately 1.73 m, which is well within the range of the second excavation. Figure 6.48 gives the corresponding nodal displacement vectors.

## 6.12 Undrained Analysis

Little mention has been made so far of the role of the dilation angle  $\psi$  on the calculation of collapse loads in Mohr–Coulomb materials. The reason is that the dilation angle governs volumetric strains during plastic yield and will have little influence on collapse loads in relatively unconfined problems.



**Figure 6.47** Excavation height vs. vertical displacement at node 31 from Program 6.10 example

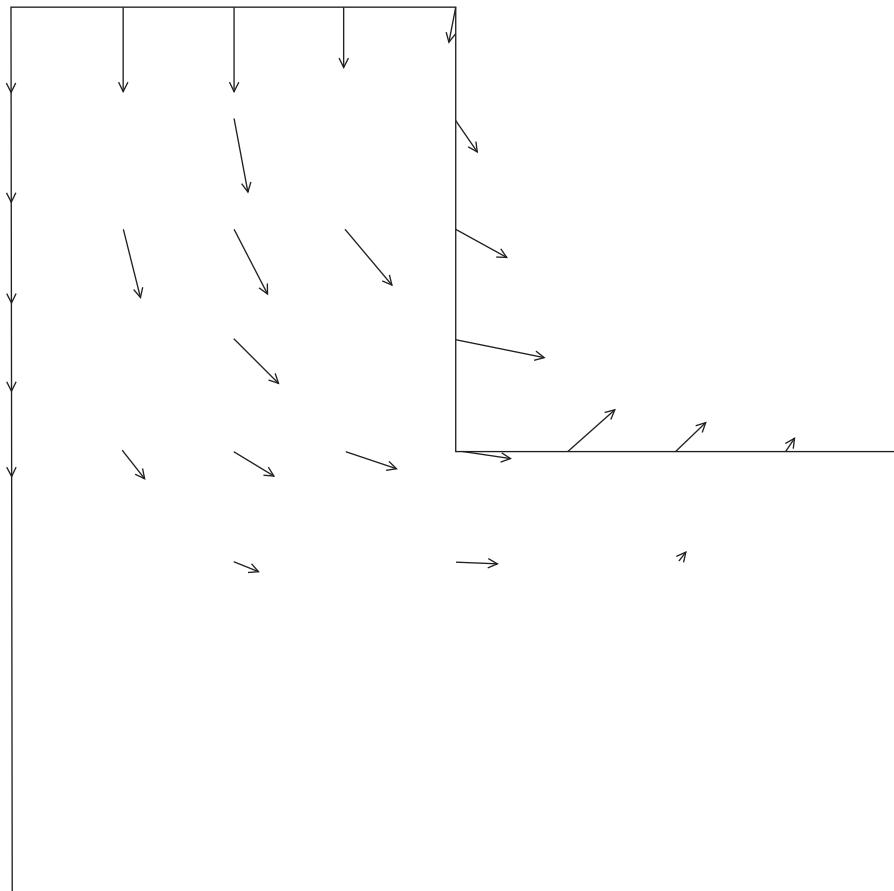
Saturated soils are two-phase particulate materials in which the voids between the particles are full of water. Under laboratory conditions, drainage may be prohibited, or alternatively, the permeability of the material may be sufficiently low and/or the loads applied so quickly that pore water pressures generated have no time to dissipate during the time scale of the analysis.

In the case of undrained clays that have soft soil skeletons, the shear strength appears to be independent of confining pressure and given by an undrained shear strength  $c_u$  ( $\phi_u = 0$ ). In such materials, the von Mises or Tresca failure criterion can be successfully applied, as was demonstrated in Program 6.1.

In the case of saturated soils with hard skeletons, such as dense quartz sand, shear stresses will tend to cause dilation which will be resisted by tensile water pressures in the voids of the soil. In turn, the effective stresses between particles will rise and, in a frictional material, the shear stresses necessary to cause failure will also rise. Thus, a dense sand, far from exhibiting a constant shear strength when sheared undrained, exhibits ever-increasing strength. In reality, a finite shear strength is eventually reached, due either to grain crushing or pore fluid cavitation.

To perform analyses of this type it is necessary to separate stresses into pore water pressures (isotropic) and effective inter-particle stresses (isotropic + shear). Such a treatment has already been described in Chapter 2 in terms of time-dependent ‘consolidation’ properties of two-phase materials (Biot’s poroelastic theory) and programs to deal with this will be found in Chapter 9. However, the undrained problem pertaining at the beginning of the Biot process is so important in soil mechanics that it merits special treatment.

Naylor (1974) described a method of separating the stresses into pore pressures and effective stresses. The method uses as its basis the concept of effective stress in matrix



**Figure 6.48** Displacement vectors at failure when excavation height reached 2 m from Program 6.10 example

notation; thus

$$\{\sigma\} = \{\sigma'\} + \{\mathbf{u}\} \quad (6.76)$$

where  $\{\sigma\}$  is the total stress,  $\{\sigma'\}$  the effective stress and  $\{\mathbf{u}\}$  the pore pressure.

The elastic stress-strain relationships can be written as

$$\{\sigma'\} = [\mathbf{D}']\{\epsilon\} \quad (6.77)$$

where

$$\{\mathbf{u}\} = [\mathbf{D}_u]\{\epsilon\} \quad (6.78)$$

Combining these equations gives

$$\{\sigma\} = [\mathbf{D}]\{\epsilon\} \quad (6.79)$$

where

$$[\mathbf{D}] = [\mathbf{D}'] + [\mathbf{D}_u] \quad (6.80)$$

The matrix  $[\mathbf{D}']$  is the familiar elastic stress–strain matrix in terms of effective Young's modulus  $E'$  and Poisson's ratio  $\nu'$  from (2.81). The matrix  $[\mathbf{D}_u]$  contains the apparent bulk modulus of the fluid  $K_e$  as

$$[\mathbf{D}_u] = \begin{bmatrix} K_e & K_e & 0 & K_e \\ K_e & K_e & 0 & K_e \\ 0 & 0 & 0 & 0 \\ K_e & K_e & 0 & K_e \end{bmatrix} \quad (6.81)$$

assuming that the third row and column correspond to the shear terms in a two-dimensional plane-strain analysis.

To implement this method in the programs described in this chapter, it is necessary to form the global stiffness matrix using the total stress–strain matrix  $[\mathbf{D}]$ , while effective stresses for use in the failure function are computed from total strains using the effective stress–strain matrix  $[\mathbf{D}']$ . Pore pressures are simply computed from

$$\{\mathbf{u}\} = K_e(\{\boldsymbol{\epsilon}_r\} + \{\boldsymbol{\epsilon}_z\} + \{\boldsymbol{\epsilon}_\theta\}) \begin{Bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{Bmatrix} \quad (6.82)$$

assuming an axisymmetric analysis.

For relatively large values of  $K_e$ , the analysis is insensitive to the exact magnitude of  $K_e$ . For axisymmetric analyses, Griffiths (1985) defined the dimensionless group

$$\beta_t = \frac{(1 - 2\nu')K_e}{E'} \quad (6.83)$$

and showed that for an undrained triaxial test in a non-dilative material ( $\psi = 0$ ), consolidated at a cell pressure of  $\sigma_3$ , the deviator stress at failure would be given by

$$D_f = \frac{\sigma_3(K_p - 1)(3\beta_t + 1)}{(K_p + 2)\beta_t + 1} \quad (6.84)$$

where  $K_p = \tan^2(45^\circ + \phi'/2)$ .

In the limit as  $\beta_t \rightarrow \infty$ , this expression tends to

$$D_f = \frac{3\sigma_3(K_p - 1)}{(K_p + 2)} \quad (6.85)$$

although for practical purposes, undrained behaviour is essentially captured for  $\beta_t \geq 20$ .

### **Program 6.11 Axisymmetric ‘undrained’ strain of an elastic–plastic (Mohr–Coulomb) solid using 8-node rectangular quadrilaterals. Viscoplastic strain method**

```
PROGRAM p611
!-----
! Program 6.11 Axisymmetric ‘undrained’ strain of an elastic-plastic
! (Mohr-Coulomb) solid using 8-node rectangular
! quadrilaterals. Viscoplastic strain method.
!-----
```

```

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,incs,iters,iy,k,limit,ndim=2,ndof=16,nels, &
neq,nip=4,nlen,nn,nod=8,nr,nst=4,nxe,nye
REAL(iwp)::bulk,c,cons,det,dq1,dq2,dq3,dsbar,dt,d4=4.0_iwp, &
d180=180.0_iwp,e,f,lode_theta,one=1.0_iwp,phi,pi,presc,psi,sigm,snph, &
penalty=1.e20_iwp,tol,two=2.0_iwp,v,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::g(:),g_g(:,:,),g_num(:,:,),kdiag(:),nf(:,:,),no(:), &
node(:),num(:),sense(:)
REAL(iwp),ALLOCATABLE::bdylds(:,bee(:,:,),bload(:,coord(:,:)),dee(:,:,), &
der(:,:,),deriv(:,:,),devp(:,eld(:,eload(:,eps(:,erate(:, &
etensor(:,:,:),evp(:,evpt(:,:,:),flow(:,:,fun(:,gc(:,g_coord(:,:,:), &
jac(:,:,km(:,:,kv(:,loads(:,m1(:,:,m2(:,:,m3(:,:,oldis(:, &
points(:,:,pore(:,:,sigma(:,storkv(:,stress(:,tensor(:,:,), &
totd(:,weights(:,x_coords(:,y_coords(:, &
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,phi,c,psi,e,v,bulk,cons
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
x_coords(nxe+1),y_coords(nye+1),num(nod),evpt(nst,nip,nels), &
coord(nod,ndim),g_g(ndof,nels),tensor(nst,nip,nels),fun(nod), &
etensor(nst,nip,nels),dee(nst,nst),pore(nip,nels),stress(nst), &
jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),g_num(nod,nels), &
bee(nst,ndof),km(ndof,ndof),eld(ndof),eps(nst),sigma(nst),bload(ndof), &
eload(ndof),erate(nst),evp(nst),devp(nst),g(ndof),m1(nst,nst), &
m2(nst,nst),m3(nst,nst),flow(nst,nst),gc(ndim))
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),bdylds(0:neq),oldis(0:neq),totd(0:neq))
!-----loop the elements to find global arrays sizes-----
kdiag=0
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I7))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----add fluid bulk modulus effective dee matrix-----
CALL deemat(dee,e,v); pi=ACOS(-one)
DO i=1,nst; DO k=1,nst; IF(i/=3.AND.k/=3)dee(i,k)=dee(i,k)+bulk
    END DO; END DO; snph=SIN(phi*pi/d180)
dt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph*snph))
CALL sample(element,points,weights); kv=zero; tensor=zero; etensor=zero
!-----element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i)
        CALL bee8(bee,coord,points(i,1),points(i,2),det)
        gc=MATMUL(fun,coord); bee(4,1:ndof-1:2)=fun(:)/gc(1)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)*gc(1)
        tensor(1:2,i,iel)=cons; tensor(4,i,iel)=cons
    END DO
END DO

```

```

END DO gauss_pts_1; CALL fsparv(kv,km,g,kdiag)
END DO elements_2
!-----read displacement data and factorise equations---
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),no(fixed_freedoms),&
        storkv(fixed_freedoms))
    READ(10,*)(node(i),sense(i),i=1,fixed_freedoms)
    DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
    kv(kdiag(no))=kv(kdiag(no))+penalty; storkv=kv(kdiag(no))
END IF; CALL sparin(kv,kdiag); CALL deemat(dee,e,v)
!-----displacement increment loop-----
READ(10,*)tol,limit,incs,presc
WRITE(11,'(/A)')" step disp dev stress pore press iters"
oldis=zero; totd=zero
disp_incs: DO iy=1,incs
    iters=0; bdylds=zero; evpt=zero
!-----plastic iteration loop-----
its: DO
    iters=iters+1
    WRITE(*,'(A,E11.3,A,I4)')" displacement",iy*presc," iteration",iters
    loads=zero; loads(no)=storkv*presc; loads=loads+bdylds
    CALL spabac(kv,loads,kdiag)
!-----check plastic convergence-----
    CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
!-----go round the Gauss points -----
elements_3: DO iel=1,nels
    bload=zero; num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    g=g(:,iel); eld=loads(g)
    gauss_pts_2: DO i=1,nip
        CALL shape_fun(fun,points,i)
        CALL bee8(bee,coord,points(i,1),points(i,2),det)
        gc=MATMUL(fun,coord); bee(4:1:ndof-1:2)=fun(:)/gc(1)
        eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel); sigma=MATMUL(dee,eps)
        stress=sigma+tensor(:,i,iel)
        CALL invar(stress,sigm,dsbar,lode_theta)
    !-----check whether yield is violated-----
    CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
    IF(f>=zero)THEN
        CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
        CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
        erate=MATMUL(flow,stress); evp=erate*dt
        evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
        eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)*gc(1)
    END IF
!-----update the Gauss Point stresses and calculate pore pressures-----
    IF(converged.OR.iters==limit)THEN
        tensor(:,i,iel)=stress
        etensor(:,i,iel)=etensor(:,i,iel)+eps+evpt(:,i,iel)
        pore(i,iel)=(etensor(1,i,iel)+etensor(2,i,iel)+&
            etensor(4,i,iel))*bulk
    END IF
END DO gauss_pts_2
!-----compute the total bodyloads vector -----
bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
END DO elements_3; IF(converged.OR.iters==limit)EXIT
END DO its; totd=totd+loads
WRITE(11,'(I5,3E12.4,I5)')iy,totd(no(1)),dsbar, pore(1,1),iters

```

```

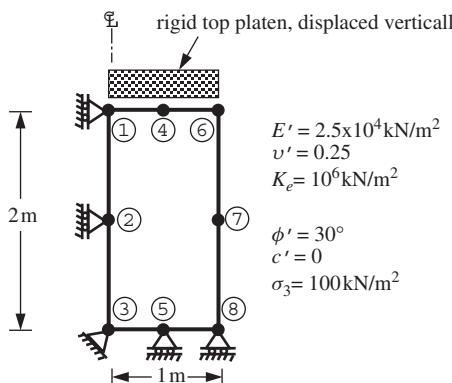
IF(iters==limit)EXIT
END DO disp_incs
CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p611

```

The example shown in Figure 6.49 represents a single axisymmetric 8-node element subjected to vertical compressive displacement increments along its top face.

The analysis is of a ‘CU’ triaxial test, in which the sample has been consolidated under a cell pressure of  $\sigma_3 = 100 \text{ kN/m}^2$ , followed by undrained axial loading. In order to compute pore pressures during undrained loading, it is necessary to update strains (*etensor*) as well as stresses after each increment. The pore pressure is also computed from equation (6.82).

This program assumes a homogeneous material described by a Mohr–Coulomb failure criterion. In addition to the usual shear strength, dilation and elastic parameters, the data file must provide the apparent fluid bulk modulus *bulk* and the initial consolidation stress *cons*.



```

nxe nye
1 1

prop(phi,c,psi,e,v,ke)
30.0 0.0 0.0 2.5e4 0.25 1.0e6

cons
-100.0

x_coords y_coords
0.0 1.0
0.0 -2.0

nr, (k,nf(:,k),i=1,nr)
5
1 0 1 2 0 1 3 0 0 5 1 0 8 1 0

fixed Freedoms, (node(i),sense(i),i=1,fixed Freedoms)
3
1 2 4 2 6 2

tol limit incs presc
0.0001 50 8 -2.0e-3

```

**Figure 6.49** Mesh and data for Program 6.11 example

There are 10 equations and the skyline storage is 55

step	disp	dev stress	pore press	iters
1	-0.2000E-02	0.2990E+02	-0.9804E+01	2
2	-0.4000E-02	0.5980E+02	-0.1961E+02	2
3	-0.6000E-02	0.8971E+02	-0.2941E+02	2
4	-0.8000E-02	0.1196E+03	-0.3922E+02	2
5	-0.1000E-01	0.1209E+03	-0.3965E+02	4
6	-0.1200E-01	0.1210E+03	-0.3966E+02	4
7	-0.1400E-01	0.1210E+03	-0.3966E+02	4
8	-0.1600E-01	0.1210E+03	-0.3966E+02	4

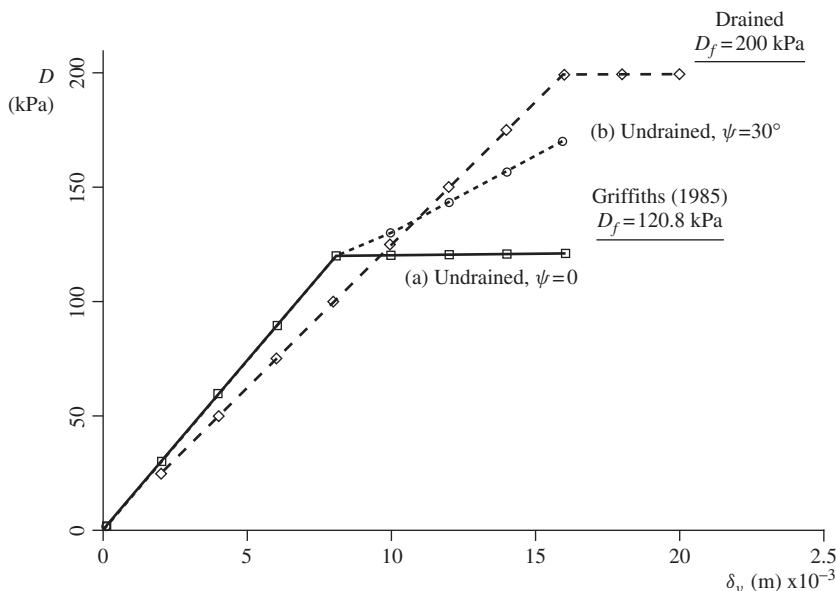
(a)  $\psi=0$

There are 10 equations and the skyline storage is 55

step	disp	dev stress	pore press	iters
1	-0.2000E-02	0.2990E+02	-0.9804E+01	2
2	-0.4000E-02	0.5980E+02	-0.1961E+02	2
3	-0.6000E-02	0.8971E+02	-0.2941E+02	2
4	-0.8000E-02	0.1196E+03	-0.3922E+02	2
5	-0.1000E-01	0.1299E+03	-0.2960E+02	3
6	-0.1200E-01	0.1439E+03	-0.2363E+02	3
7	-0.1400E-01	0.1570E+03	-0.1683E+02	3
8	-0.1600E-01	0.1703E+03	-0.1022E+02	3

(b)  $\psi=30^\circ$

**Figure 6.50** Results from Program 6.11 example with (a)  $\psi = 0$  and (b)  $\psi = 30^\circ$



**Figure 6.51** Axial displacement vs. deviator stress for drained and undrained triaxial loading from Program 6.11 example with (a)  $\psi = 0$  and (b)  $\psi = 30^\circ$

After the effective stress–strain matrix has been augmented by the fluid bulk modulus according to (6.80), the global stiffness matrix is formed in the usual way. Prescribed axial displacement increments are then applied to the top of the element using the ‘penalty’ method as used previously, for example in Programs 6.3 and 6.5.

Just before the displacement increment loop begins, the subroutine `deemat` is called to form the effective stress–strain matrix. The data shown in Figure 6.49 is for an undrained ‘sand’ with the following properties:

$$\begin{aligned}\phi' &= 30^\circ \\ c' &= 0 \\ E' &= 2.5 \times 10^4 \text{ kN/m}^2 \\ v' &= 0.25 \\ K_e &= 10^6 \text{ kN/m}^2\end{aligned}$$

The triaxial specimen has been consolidated under a compressive cell pressure of  $100 \text{ kN/m}^2$  (`cons=-100.0`) before undrained axial loading commences.

The output of two analyses is presented in Figure 6.50. In analysis (a),  $\psi = 0$  and in analysis (b),  $\psi = 30^\circ$ . As expected, the inclusion of dilation has a considerable impact on the response in this ‘confined’ problem. The deviator stress vs. vertical deflection has been plotted for both undrained cases in Figure 6.51 together with the drained result obtained by setting  $K_e = 0$ . In case (a), where there is no plastic volume change ( $\psi = 0$ ), the deviator stress reaches a peak of  $121 \text{ kN/m}^2$ , which is in close agreement with the closed-form solution of  $120.8 \text{ kN/m}^2$  (Griffiths, 1985) given by (6.84) for this problem with  $\beta_t = 20$ .

The deviator stress at failure in case (a) is significantly smaller than the drained value of  $200 \text{ kN/m}^2$  due to the compressive pore pressures generated during elastic compression. Case (b), which includes an associated flow rule ( $\psi = \phi' = 30^\circ$ ), shows no sign of failure due to the tendency for dilation. In this case, the pore pressures continue to fall and the deviator stress continues to rise. This trend would continue indefinitely unless some additional criterion (e.g., cavitation or particle crushing) was introduced. It may also be noted from Figure 6.51 that the undrained response is slightly stiffer than the drained response in the elastic range.

While the presence of dilation has a very significant influence on undrained behaviour, it has little influence on the deviator stress at failure in the drained case.

## **Program 6.12 Three-dimensional strain analysis of an elastic–plastic (Mohr–Coulomb) slope using 20-node hexahedra. Gravity loading. Viscoplastic strain method**

```
PROGRAM p612
!-----
! Program 6.12 Three-dimensional strain analysis of an elastic-plastic
! (Mohr-Coulomb) slope using 20-node hexahedra. Viscoplastic
! strain method.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15),npri=1
INTEGER::i,iel,ifix,iters,iy,limit,ndim=3,ndof=60,nels,neq,nip=8,nlen,nn,&
nod=20,nodof=3,nprops=6,np_types,nsrf,nst=6,nx1,nx2,ny1,ny2,nze
```

```

REAL(iwp)::cf,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp,dtim=1.0_iwp,d1,
d4=4.0_iwp,d180=180.0_iwp,e,f,fmax,h1,h2,lode_theta,one=1.0_iwp,phi,
phif,pi,psi,psif,sigm,snph,start_dt=1.e15_iwp,s1,tnph,tnps,tol,
two=2.0_iwp,v,w1,w2,zero=0.0_iwp
CHARACTER(LEN=80)::argv,element='hexahedron';
LOGICAL::converged,solid=.TRUE.

!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_(:, :, ),g_num(:, :, ),kdiag(:,nf(:, :, ), &
num(:)

REAL(iwp),ALLOCATABLE::bdylds(:,bee(:, :, ),bload(:,coord(:, :, ),dee(:, :, ), &
der(:, :, ),deriv(:, :, ),devp(:,eld(:,eload(:,eps(:,erate(:,evp(:, &
evpt(:, :, :,flow(:, :, ),fun(:,gravlo(:,g_coord(:, :, ),jac(:, :, ),km(:, :, ), &
kv(:,loads(:,m1(:, :, ),m2(:, :, ),m3(:, :, ),oldis(:,points(:, :, ),prop(:, :, ), &
sigma(:,srf(:,weights(:,

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
!---(ifix=1) smooth-smooth; (ifix=2) rough-smooth; (ifix=3) rough-rough---
READ(10,*)w1,s1,w2,h1,h2,d1,nx1,nx2,ny1,ny2,nze,ifix,np_types
nels=(nx1*ny1+ny2*(nx1+nx2))*nze
nn=((3*(ny1+ny2)+2)*nx1+2*(ny1+ny2)+1+(3*ny2+2)*nx2)*(1+nze) + &
((ny1+ny2+1)*(nx1+1)+(ny2+1)*nx2)*nze
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn), &
num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod), &
g_g(ndof,nels),jac(ndim,ndim),der(ndim,nod),etype(nels), &
deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof), &
eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst), &
devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst), &
prop(nprops,np_types))

READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
CALL emb_3d_bc(ifix,nx1,nx2,ny1,ny2,nze,nf); neq=MAXVAL(nf)
ALLOCATE(loads(0:neq),bdylds(0:neq),oldis(0:neq),gravlo(0:neq),kdiag(neq))
!-----loop the elements to find global arrays sizes-----
loads=zero; kdiag=0; pi=ACOS(-one)
elements_1: DO iel=1,nels
    CALL emb_3d_geom(iel,nx1,nx2,ny1,ny2,nze,w1,s1,w2,h1,h2,d1,coord,num)
    g_num(:,iel)=num; CALL num_to_g(num,nf,g)
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(A,I7,A,I8)') &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
kv=zero; oldis=zero; gravlo=zero; CALL sample(element,points,weights)
!-----element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    g=g_g(:,iel); km=zeros; eld=zeros
    gauss_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        eld(2:ndof-1:3)=eld(2:ndof-1:3)+fun(:)*det*weights(i)
    END DO gauss_pts_1
    CALL fsparv(kv,km,g,kdiag); gravlo(g)=gravlo(g)-eld*prop(4,etype(iel))
END DO elements_2
!-----factorise equations-----

```

```

CALL sparin(kv,kdiag)
!-----trial strength reduction factor loop-----
READ(10,*)tol,limit,nsrf; ALLOCATE(srf(nsrf)); READ(10,*)srf
CALL mesh_ensi(argv,nlen,g_coord,g_num,element,etype,nf,loads(1:),      &
               nsrf,npri,dtim,solid)
WRITE(11,'(/A)')"      srf      max disp   iters"
srf_trials: DO iy=1,nsrf
    dt=start_dt
    DO i=1,np_types
        phi=prop(1,i); tnph=TAN(phi*pi/d180); phif=ATAN(tnph/srf(iy))
        snph=sin(phif); e=prop(5,i); v=prop(6,i)
        ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
    END DO
    iters=0; bdylds=zero; evpt=zero
!-----plastic iteration loop-----
its: DO
    fmax=zero; iters=iters+1; loads=gravlo+bdylds
    CALL spabac(kv,loads,kdiag); loads(0)=zero
!-----check plastic convergence-----
    CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
    IF(converged.OR.iters==limit)bdylds=zero
!-----go round the Gauss Points -----
elements_3: DO iel=1,nels
    phi=prop(1,etype(iel)); tnph=TAN(phi*pi/d180)
    phif=ATAN(tnph/srf(iy))*d180/pi; psi=prop(3,etype(iel))
    tnps=TAN(psi*pi/d180); psif=ATAN(tnps/srf(iy))*d180/pi
    cf=prop(2,etype(iel))/srf(iy); e=prop(5,etype(iel))
    v=prop(6,etype(iel)); CALL deemat(dee,e,v); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g(:,iel); eld=loads(g)
    bload=zero
    gauss_points_2: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
        eps=eps-evpt(:,i,iel); sigma=MATMUL(dee,eps)
        CALL invar(sigma,sigm,dsbar,lode_theta)
    !-----check whether yield is violated-----
        CALL mocouf(phif,cf,sigm,dsbar,lode_theta,f); IF(f>fmax)fmax=f
        IF(converged.OR.iters==limit)THEN; devp=sigma; ELSE
            IF(f>=zero.OR.(converged.OR.iters==limit))THEN
                CALL mocouq(psif,dsbar,lode_theta,dq1,dq2,dq3)
                CALL formm(sigma,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
                erate=MATMUL(flow,sigma); evp=erate*dt
                evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
            END IF
        END IF
        IF(f>=zero.OR.(converged.OR.iters==limit))THEN
            eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
        END IF
    END DO gauss_points_2
!-----compute the total bodyloads vector-----
    bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
END DO elements_3
WRITE(*,'(A,F7.2,A,I4,A,F8.3)')      &
    "      srf",srf(iy),"  iteration",iters,"  F_max",fmax
IF(converged.OR.iters==limit)EXIT
END DO its
WRITE(11,'(F7.2,E12.4,I5)')srf(iy),MAXVAL(ABS(loads)),iters

```

```

CALL dismsh_ensi(argv,nlen,iy,nf,oldis(1:))
IF(iters==limit)EXIT
END DO srf_trials
STOP
END PROGRAM p612

```

This program demonstrates 3D plasticity analysis using 20-node hexahedral elements with ‘reduced’ ( $nip=8$ ) integration (Griffiths and Marquez 2007). The example is of a simple 3D slope stability analysis, and the program is very similar to its 2D counterpart Program 6.4. Only three additional inputs are required as compared with the data for Program 6.4. The first is `ifix`, which fixes the front and back faces of the mesh (in the  $z$ -direction) to either ‘rough’ or ‘smooth’. When  $ifix=1$ , both boundaries are smooth, and if the slope is homogeneous the analysis essentially reduces to plane strain; when  $ifix=2$ , the front face is rough and the back face smooth, implying a line of symmetry along the centre of the embankment and when  $ifix=3$ , both boundaries are rough, enabling a full 3D analysis of a slope that may have non-uniform and non-symmetric properties in the crest ( $z$ -) direction. The second new input parameter is `nze`, which defines the number of slices of elements required in the  $z$ -direction and the third is `d1`, which represents the depth of the slope in the  $z$ -direction.

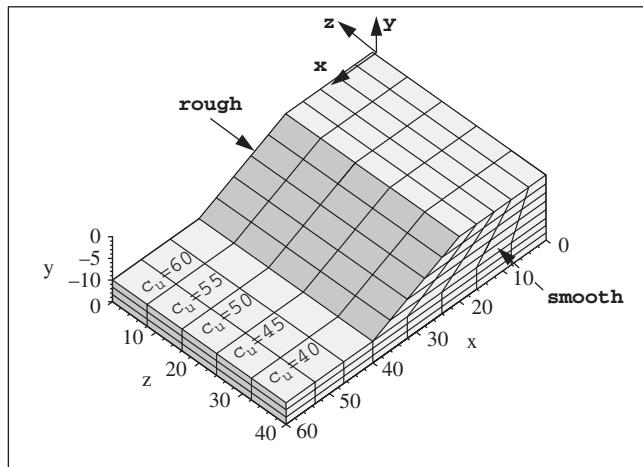
Two new subroutines, `emb_3d_bc` and `emb_3d_geom`, are introduced to generate, respectively, the nodal freedom array `nf` and the nodal coordinates and element node numbering `g_coord` and `g_num`. For the purposes of this example, the subroutines create a rather simple 3D geometry in which the 2D cross-section is extrapolated uniformly in the  $z$ -direction. Users are invited to introduce their own mesh-generation techniques to study more realistic 3D geometries.

The slope and data to be considered are shown in Figure 6.52. The figure shows a 2:1 slope with a height of 10 m and a foundation depth factor of  $D = 1.5$ . The depth of the mesh in the  $z$ -direction is 40 m  $d1=40$ , and the boundary condition parameter is set to  $ifix=2$ , implying a line of symmetry at the  $z = 40$  m plane (rough–smooth), so the ‘actual’ embankment has a depth of 80 m.

Three-dimensional analysis involves very significant storage requirements compared with 2D, so the example involves a quite coarse mesh with five slices of elements in the  $z$ -direction  $nze=5$ . In this case the slope is assumed to have a linearly varying undrained strength varying from 60 kN/m<sup>2</sup> at the abutment ( $z = 0.0$ ) to 40 kN/m<sup>2</sup> at the centreline ( $z = -40.0$ ). There are thus five property types (`np_types=5`) in the data file, one for each slice.

The `etype` data maps the properties onto the elements, which are numbered first in the  $x$ -direction, then in the negative  $y$ -direction (top to bottom) and finally in the negative  $z$ -direction. The tolerance `tol` and iteration ceiling `limit` are set as usual, followed by the number of trial strength reduction factors `nsrf` and the strength reduction factor values read into `srf`. The iteration ceiling has been set higher than usual at 1000 to emphasise the onset of failure.

The output from the analysis is given in Figure 6.53 and plotted in Figure 6.54. The results indicate that the factor of safety of the slope is about 1.6. Figure 6.55 shows the deformed mesh at failure (corresponding to the unconverged solution) as generated using ParaView. Slumping of the slope towards its centreline at failure is clearly seen. It may be noted that ParaView stores results following all the trial strength reduction factors, allowing the option of animations.



```

w1    s1    w2    h1    h2    d1
20.0  20.0  20.0  10.0  5.0   40.0

nx1   nx2   ny1   ny2   nze
5      3      5      3      5

ifix   np_types
2      5

prop(phi,c,psi,gamma,e,v)
0.0 60.0  0.0 20.0  1.0e5  0.3
0.0 55.0  0.0 20.0  1.0e5  0.3
0.0 50.0  0.0 20.0  1.0e5  0.3
0.0 45.0  0.0 20.0  1.0e5  0.3
0.0 40.0  0.0 20.0  1.0e5  0.3

etype(i),i=1,nels (x then y then z)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5

tol    limit
0.0001 1000

nsrf,(srf(i),i=1,nsrf)
6
1.0 1.4 1.5 1.55 1.58 1.60

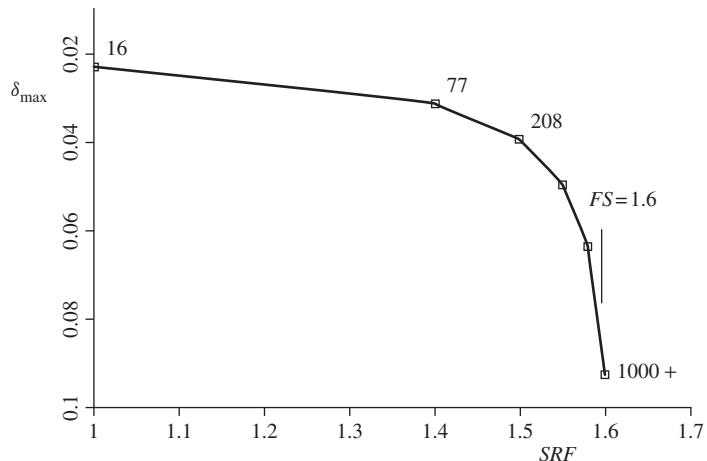
```

**Figure 6.52** Mesh and data for Program 6.12 example

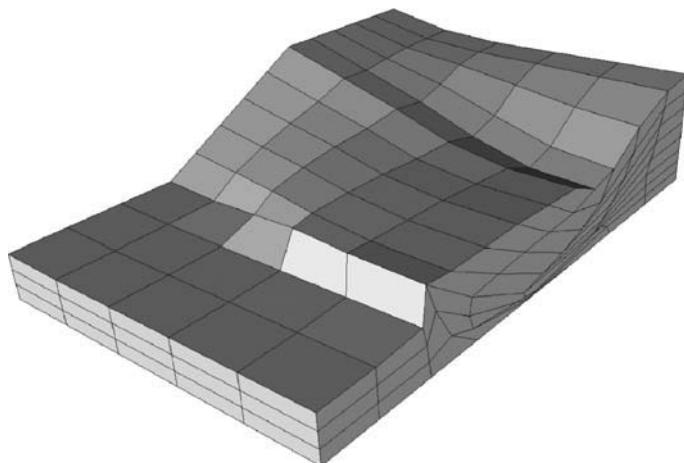
There are 2972 equations and  
the skyline storage is 1538004

srf	max disp	iters
1.00	0.2266E-01	16
1.40	0.3123E-01	77
1.50	0.3930E-01	208
1.55	0.4971E-01	380
1.58	0.6363E-01	703
1.60	0.9308E-01	1000

**Figure 6.53** Results from Program 6.12 example



**Figure 6.54** Strength reduction factor vs. maximum displacement from Program 6.12 example



**Figure 6.55** Deformed mesh at failure from Program 6.12 example

**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**

```

PROGRAM p613
!-----
! Program 6.13 Three-dimensional strain analysis of an elastic-plastic
! (Mohr-Coulomb) slope using 20-node hexahedra. Viscoplastic
! strain method. No global stiffness matrix assembly.
! Diagonally preconditioned conjugate gradient solver.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,cg_tot,i,iel,ifix,iters,iy,k,limit,ndim=3,      &
ndof=60,nels,neq,nip=8,nn,nod=20,nodof=3,nprops=6,np_types,nsrf,nst=6,      &
nx1,nx2,ny1,ny2,nze,nlen
REAL(iwp)::alpha,beta,cf,cg_tol,ddt,det,dq1,dq2,dq3,dsbar,dt=1.0e15_iwp, &
d1,d4=4.0_iwp,d180=180.0_iwp,e,f,fmax,h1,h2,lode_theta,one=1.0_iwp,phi, &
phif,pi,psi,psif,sigm,snph,start_dt=1.e15_iwp,s1,tnph,tnps,tol,           &
two=2.0_iwp,up,v,w1,w2,zero=0.0_iwp
CHARACTER(LEN=80)::argv,element='hexahedron'
LOGICAL::converged,cg_converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:,),g_g(:,,:),g_num(:,,:),nf(:,,:),num(:)
REAL(iwp),ALLOCATABLE::bdyls(:,),bee(:,,:),bload(:, ),coord(:,,:),d(:, ),
dee(:, ),der(:, ),deriv(:, ),devp(:, ),diag_precon(:, ),eld(:, ),eload(:, ),      &
eps(:, ),erate(:, ),evp(:, ),evpt(:, :, ),flow(:, ),fun(:, ),gravlo(:, ),      &
g_coord(:, ),jac(:, ),km(:, ),loads(:, ),m1(:, ),m2(:, ),m3(:, ),      &
oldis(:, ),p(:, ),points(:, ),prop(:, ),sigma(:, ),srf(:, ),storkm(:, :, ),u(:, ),      &
weights(:, ),x(:, ),xnew(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
!---(ifix=1) smooth-smooth; (ifix=2) rough-smooth; (ifix=3) rough-rough---
READ(10,*)w1,s1,w2,h1,h2,d1,nx1,nx2,ny1,ny2,nze,ifix,                      &
cg_tol,cg_limit,np_types
nels=(nx1*ny1+ny2*(nx1+nx2))*nze
nn=(3*(ny1+ny2)+2)*nx1+2*(ny1+ny2)+1+(3*ny2+2)*nx2)*(1+nze)+          &
(ny1+ny2+1)*(nx1+1)+(ny2+1)*nx2)*nze
ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),      &
num(nod),dee(nst,nst),evpt(nst,nip,nels),coord(nod,ndim),fun(nod),      &
g_g(ndof,nels),jac(ndim,ndim),der(ndim,nod),etype(nels),      &
deriv(ndim,nod),g_num(nod,nels),bee(nst,ndof),km(ndof,ndof),eld(ndof),      &
eps(nst),sigma(nst),bload(ndof),eload(ndof),erate(nst),evp(nst),      &
devp(nst),g(ndof),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst),      &
prop(nprops,np_types),storkm(ndof,ndof,nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
CALL emb_3d_bc(ifix,nx1,nx2,ny1,ny2,nze,nf); neq=MAXVAL(nf)
WRITE(11,'(A,I7,A)')" There are",neq," equations"
ALLOCATE(loads(0:neq),bdyls(0:neq),oldis(0:neq),gravlo(0:neq),p(0:neq), &
x(0:neq),xnew(0:neq),u(0:neq),diag_precon(0:neq),d(0:neq))
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    CALL emb_3d_geom(iel,nx1,nx2,ny1,ny2,nze,w1,s1,w2,h1,h2,d1,coord,num)
    g_num(:,iel)=num; CALL num_to_g(num,nf,g)

```

```

g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1
pi=ACOS(-one); oldis=zero; gravlo=zero; p=zero; xnew=zero
diag_precon=zero; CALL sample(element,points,weights)
!-----element stiffness integration, storage and preconditioner-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(5,etype(iel)),prop(6,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    g=g_g(:,iel); km=zeros; eld=zeros
    gauss_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        eld(2:ndof-1:3)=eld(2:ndof-1:3)+fun(:)*det*weights(i)
    END DO gauss_pts_1
    storkm(:,:,iel)=km
    DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
    gravlo(g)=gravlo(g)-eld*prop(4,etype(iel))
END DO elements_2
diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
!-----trial strength reduction factor loop-----
READ(10,*)tol,limit,nsrf; ALLOCATE(srf(nsrf))
READ(10,*)srf
WRITE(11,'(/A)')"      srf      max disp   iters      cg iters/plastic iter"
srf_trials: DO iy=1,nsrf
    dt=start_dt
    DO i=1,np_types
        phi=prop(1,i); tnph=TAN(phi*pi/d180); phif=ATAN(tnph/srf(iy))
        snph=SIN(phif); e=prop(5,i); v=prop(6,i)
        ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
    END DO
    iters=0; bdylds=zero; evpt=zero; cg_tot=0
!-----plastic iteration loop-----
its: DO
    fmax=zero; iters=iters+1; loads=gravlo+bdylds; d=diag_precon*loads
    p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
pcg: DO
    cg_iters=cg_iters+1; u=zero
    elements_3: DO iel=1,nels
        CALL deemat(dee,prop(2,etype(iel)),prop(3,etype(iel)))
        g=g_g(:,iel); km=storkm(:,:,iel); u(g)=u(g)+MATMUL(km,p(g))
    END DO elements_3
    up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u)
    xnew=x+p*alpha; loads=loads-u*alpha; d=diag_precon*loads
    beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
    CALL checon(xnew,x,cg_tol,cg_converged)
    IF(cg_converged.OR.cg_iters==cg_limit)EXIT
    END DO pcg
    cg_tot=cg_tot+cg_iters; loads=xnew; loads(0)=zero
!-----check plastic convergence-----
CALL checon(loads,oldis,tol,converged); IF(iters==1)converged=.FALSE.
    IF(converged.OR.iters==limit)bdylds=zero
!-----go round the Gauss Points -----
elements_4: DO iel=1,nels
    phi=prop(1,etype(iel)); tnph=TAN(phi*pi/d180)
    phif=ATAN(tnph/srf(iy))*d180/pi; psi=prop(3,etype(iel))

```

```

tnps=TAN(psi*pi/d180); psif=ATAN(tnps/srf(iy))*d180/pi
cf=prop(2,etype(iel))/srf(iy); e=prop(5,etype(iel))
v=prop(6,etype(iel)); CALL deemat(dee,e,v); num=g_num(:,iel)
coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g)
bload=zeros
gauss_points_2: DO i=1,nip
    CALL shape_der(der,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
    eps=MATMUL(bee,eld); eps=eps-evpt(:,i,iel)
    sigma=MATMUL(dee,eps); CALL invar(sigma,sigm,dsbar,lode_theta)
!-----check whether yield is violated-----
    CALL mocouf(phif,cf,sigm,dsbar,lode_theta,f); IF(f>fmax) fmax=f
    IF(converged.OR.iters==limit)THEN; devp=sigma; ELSE
        IF(f>=zero.OR.(converged.OR.iters==limit))THEN
            CALL mocouq(psif,dsbar,lode_theta,dq1,dq2,dq3)
            CALL formm(sigma,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
            erate=MATMUL(flow,sigma); evp=erate*dt
            evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
        END IF
    END IF
    IF(f>=zero.OR.(converged.OR.iters==limit))THEN
        eload=MATMUL(devp,bee); bload=bload+eload*det*weights(i)
    END IF
END DO gauss_points_2
!-----compute the total bodyloads vector-----
bdylds(g)=bdylds(g)+bload; bdylds(0)=zero
END DO elements_4
WRITE(*,'(A,F7.2,A,I4,A,F8.3)') &
    " srf",srf(iy)," iteration",iters," F_max",fmax
IF(converged.OR.iters==limit)EXIT
END DO its
WRITE(11,'(F7.2,E12.4,I5,F17.2)') &
    srf(iy),MAXVAL(ABS(loads)),iters,REAL(cg_tot)/REAL(iters)
IF(iters==limit)EXIT
END DO srf_trials
STOP
END PROGRAM p613

```

The final program in this chapter repeats the analysis of Program 6.12, using a preconditioned conjugate gradient solver involving no mesh assembly. The rather coarse mesh demonstrated in the previous example still required 1,538,004 locations in the vector *kv* in order to store the stiffness matrix using a skyline strategy. Even quite modest 3D meshes require arrays with millions of locations in order to store the global matrices.

The ‘element-by-element’ approach made possible with pcg solvers enables large 3D analyses to be performed on computers with modest core memory availability. Furthermore, the pcg approach is highly amenable to exploitation on computers with parallel architecture, as will be demonstrated in detail in Chapter 12.

Program 6.13 involves no new variables and can be considered to be an extension of Program 6.2, in which the pcg technique was first introduced in a plasticity analysis.

The example shown in Figure 6.56 is the same slope as was demonstrated previously with Program 6.12. The two additional data values required are *pcg\_tol* set to 0.0001, and *pcg\_limit* set to 500. The output shown in Figure 6.57 is essentially identical to those obtained using the direct solver.

**Figure 6.56** Data for Program 6.13 example

There are 2972 equations

srf	max disp	iters	cg iters/plastic iter
1.00	0.2264E-01	17	107.88
1.40	0.3119E-01	78	110.40
1.50	0.3918E-01	206	110.41
1.55	0.44959E-01	379	110.11
1.58	0.6377E-01	723	110.04
1.60	0.9431E-01	1000	110.03

**Figure 6.57** Results from Program 6.13 example

## 6.13 Glossary of Variable Names

## Scalar integers:

<code>cg_iters</code>	conjugate gradient iteration counter
<code>cg_limit</code>	conjugate gradient iteration ceiling
<code>cg_tot</code>	keeps running total of <code>cg_iters</code>
<code>enxe, enye</code>	number of $x$ - and $y$ - elements in embankment
<code>fixed_freedoms</code>	number of fixed displacements
<code>fnxe, fnye</code>	number of $x$ - and $y$ -elements in foundation
<code>i, iel</code>	simple counters
<code>ifix</code>	sets 3D slope boundary conditions
<code>ii</code>	counts the lifts

incs	number of load increments
iters	counts plastic iterations
itype	type of degeneration of quadrilateral to triangle
iy	counts load increments
iwp	SELECTED_REAL_KIND(15)
i3,i4,i5,j,jj	simple counters
k	node number
layers	number of excavation steps
lifts	number of lifts
limit	plastic iteration ceiling
lnn	keeps running total of number of nodes
loaded_nodes	number of loaded nodes
nbo2	number of elements under half rigid footing
ndim	number of dimensions
newele	number of new elements at each lift
ndof	number of degrees of freedom per element
nels	number of elements
neq	number of degrees of freedom in the mesh
nip	number of integrating points per element
nlen	maximum number of characters in data file basename
nn	number of nodes
nod	number of nodes per element
nodof	number of degrees of freedom per node
noexe	number of elements to be removed at each step
nouts	number of nodes at which output is required
nprops	number of material properties
np_types	number of different property types
nr	number of restrained nodes
nsrf	number of trial strength reduction factors
nst	number of stress (strain) terms
ntote	holds running total of number of excavated elements
nxe	number of columns elements in $x$ -direction
nx1	number of columns of elements in embankment
nx2	number of columns of elements to right of toe
nye	number of rows of elements in $y$ -direction
ny1	number of rows of elements in embankment
ny2	number of rows of elements in foundation
nze	number of slices of elements in $z$ -direction
oldele	keeps running total of number of elements
oldnn	number of nodes from previous lift

**Scalar reals:**

alpha,beta	$\alpha$ and $\beta$ from (3.22)
bot	holds several dot products
bulk	apparent fluid bulk modulus
c	cohesion
cf	factored cohesion
cg_tol	pcg convergence tolerance

**Scalar reals (continued):**

cons	consolidating stress ( $\sigma_3$ )
con1, con2	used in radial stress correction
c_e	embankment cohesion
c_f	foundation cohesion
dlam	plastic multiplier $\lambda$
ddt	used to find the critical time step
det	determinant of the Jacobian matrix
dq1	plastic potential derivative, $\partial Q / \partial \sigma_m$
dq2	plastic potential derivative, $\partial Q / \partial J_2$
dq3	plastic potential derivative, $\partial Q / \partial J_3$
dsbar	invariant, $\bar{\sigma}$
dslam	plastic multiplier increment $\Delta\lambda$
dt	critical viscoplastic time step (set initially to $10^{15}$ )
d1	depth of mesh in z-direction
d3, d4, d6, d180	set to 3.0, 4.0, 6.0 and 180.0
e	Young's modulus
end_time	stop CPU clock
e_e	Young's modulus in embankment
e_f	Young's modulus in foundation
f	value of yield function
fac	measure of yield surface overshoot ( $f$ from 6.35)
ff	holds a value of the yield function
fttol	tolerance on yield function
fmax	maximum value of yield function $f$
fnew	value of yield function after stress increment
fstiff	holds a value of the yield function
gama_e	unit weight of embankment
gama_f	unit weight of foundation
gamma	soil unit weight
h1	height of embankmemnt
h2	height of foundation
k0	"at rest" earth pressure coefficient, $K_0$
load_theta	Lode angle, $\theta$
ltol	tolerance on tloads
one	set to 1.0
ot	overturning moment
pav	earth force based on stress averaging
penalty	set to $1 \times 10^{20}$
phi	friction angle (degrees)
phif	factored friction angle
phi_e	friction angle of embankment
phi_f	friction angle of foundation
pi	set to $\pi$
pr	earth force based on nodal reactions
presc	wall displacement increment
psi	dilation angle (degrees)
psi_e	dilation angle of embankment

psi_f	dilation angle of foundation
psif	factored dilation angle
ptot	holds running total of applied pressure
pt5	set to 0.5
qq	force on element side
qs	surface surcharge
sigm	mean stress, $\sigma_m$
snph	sin of phi
start_dt	starting value of dt
start_time	start CPU clock
sx,sy,sz	deviatoric stress in x-, y- and z-directions
s0	used in line search
s1	used in line search or width of top of embankment
tloads	holds the sum of bdylds
tnph	tangent of phi
tnps	tangent of psi
tol	plastic convergence tolerance
top	holds a dot product
two	set to 2.0
up	holds dot product $\{\mathbf{R}\}_k^T \{\mathbf{R}\}_k$ from (3.22)
v	Poisson's ratio
v_e	Poisson's ratio of embankment
v_f	Poisson's ratio of foundation
w1	width of sloping section of embankment
w2	distance foundation extends beyond the toe
zero	set to 0.0

**Scalar character:**

argv	holds data file basename
element	element type ('quadrilateral')

**Scalar logicals:**

cg_converged	set to .TRUE. if pcg process has converged
converged	set to .TRUE. if plastic iterations have converged

**Dynamic integer arrays:**

etype	element property types
exele	element numbers of removed elements
g	element "steering" vector
g_g	steering vector for all elements
g_num	node numbers for all elements
kdiag	diagonal term locations
lnf	local nodal freedoms
nf	nodal freedoms
no	freedoms to be fixed
node	nodes with fixed displacement
num	element node numbers
sense	sense of freedom to be fixed

**Dynamic integer arrays (continued):**

solid	identifies “air” elements ( $=0$ for “air”, $=1$ for solid)
totex	holds element numbers for all removed elements

**Dynamic real arrays:**

acat, acatc	used in development of (6.73)
bddylds	self-equilibrating global body-loads
bddylds0	used in line search
bee	strain-displacement matrix
bload	self-equilibrating element body-loads
caflow	used in development of (6.73)
coord	element nodal coordinates
d	vector used in (3.22)
daatd	used in development of (6.73)
ddylds	global body-loads
dee	stress strain matrix
deep	corrected elastic-plastic matrix
der	shape function derivatives with respect to local coordinates
deriv	shape function derivatives with respect to global coordinates
devp	product $[\mathbf{D}^e]\{\Delta\epsilon^{vp}\}$
diag_precon	diagonal preconditioner vector
dl	holds plastic multiplier $\lambda$ for all Gauss points
dload	element body-loads
dsigma	stress increment
elastic	elastic nodal displacements
eld	element nodal displacements
eload	integrating point contribution to bload
elso	plastic stresses
eps	strain terms
erate	viscoplastic strain rate, $\{\dot{\epsilon}^{vp}\}$
etensor	holds running total of all integrating point strain terms
evp	viscoplastic strain rate increment, $\{\Delta\epsilon^{vp}\}$
evpt	holds running total of viscoplastic strains, $\{\Delta\epsilon^{vp}\}$
exc_loads	excavation loads
flow	holds $\{\partial Q/\partial\sigma\}$
fun	shape functions
gc	integrating point coordinates
gravlo	loads generated by gravity
g_coord	nodal coordinates for all elements
jac	Jacobian matrix
km	element stiffness matrix
kv	global stiffness matrix
kvc	copy of kv
loads	nodal loads and displacements
loadsr	total loads
m1	used to compute $\{\partial\sigma_m/\partial\sigma\}$

m2	used to compute $\{\partial J_2/\partial \sigma\}$
m3	used to compute $\{\partial J_3/\partial \sigma\}$
oldis	nodal displacements from previous iteration
p	“descent” vector used in (3.22)
p1	plastic $[\mathbf{D}^p]$ matrix
points	integrating point local coordinates
pore	holds running total of all integrating point pore pressures
prop	element properties
qinc	holds applied pressure increments
qinva, qinvr, qmat	used in development of (6.73)
react	global nodal reaction forces
ress, rmat	used in development of (6.73)
rload	element nodal reaction forces
sigma	stress terms
srf	trial strength reduction factors
storkm	holds element stiffness matrices
storkv	holds augmented stiffness diagonal terms
stress	stress term increments
tensor	holds running total of all integrating point stress terms
tensorl	Additional storage of integrating point stress terms
totd	holds running total of nodal displacements
totdl	holds incremental nodal displacements
totdll, totdlo	used as part of line search
tot_d	holds running total of nodal displacements
trial	trial stress
u	vector used in (3.22)
val	applied nodal load weightings
vmfl	von Mises “flow” vector
vmfla, vmflq, vmtemp	used in development of (6.73)
value	fixed values of displacements
weights	weighting coefficients
x, xnew	“old” and “new” solution vector
x_coords, y_coords	x- and y-coordinates of mesh layout

## 6.14 Exercises

1. Use Program 6.1 to estimate the bearing capacity of the surface strip footing shown in Figure 6.58 which is supported by undrained clay with a shear strength of  $c_u = 50 \text{ kN/m}^2$ .

Answer:  $q_{ult} \approx 257 \text{ kN/m}^2$

2. Repeat question 1 if the undrained shear strength increases linearly from  $20 \text{ kN/m}^2$  at ground level to  $50 \text{ kN/m}^2$  at 5 m depth.

Answer:  $q_{ult} \approx 130 \text{ kN/m}^2$

3. Use Program 6.1 to estimate the bearing capacity of the footing shown in Figure 6.59 which is at the edge of a vertical cut of undrained clay with a shear strength of  $c_u = 75 \text{ kN/m}^2$ .

Answer:  $q_{ult} = 150 \text{ kN/m}^2$

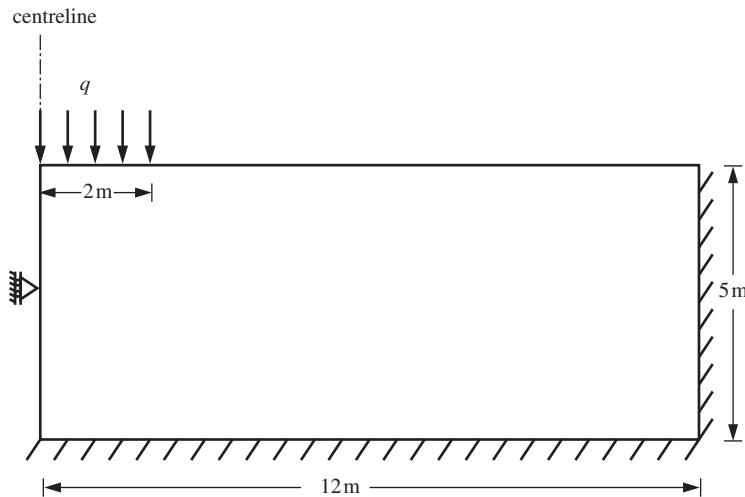


Figure 6.58

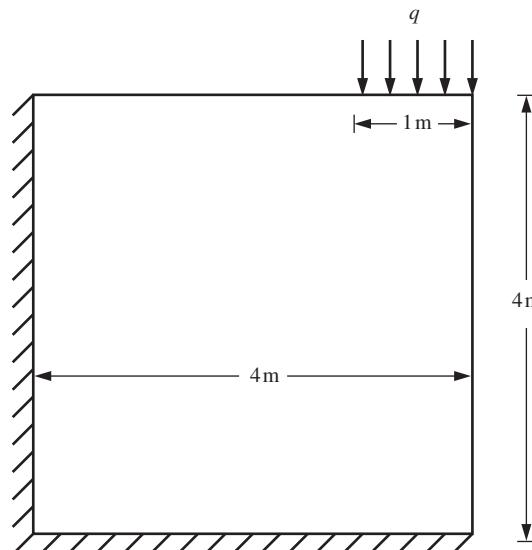


Figure 6.59

4. Modify the data shown in Figure 6.15 to estimate the bearing capacity of the same rigid strip footing with modified soil properties given by  $\phi' = 25^\circ$ ,  $c' = 20 \text{ kPa}$ ,  $\gamma = 17.5 \text{ kN/m}^3$ , with a surcharge of  $q = 10 \text{ kPa}$  acting at the ground surface.

Answer:  $q_{ult} \approx 755 \text{ kN/m}^2$

5. Use Program 6.4 to estimate the factor of safety of the slope shown in Figure 6.60.

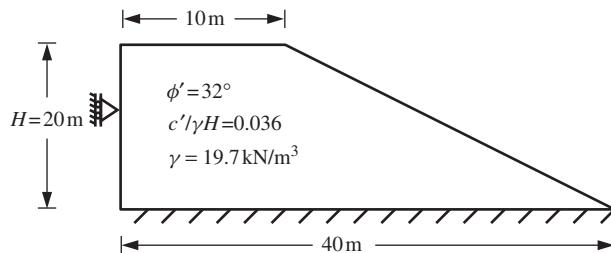


Figure 6.60

Answer:  $F \approx 1.45$

6. Use Program 6.4 to repeat the previous analysis if a second layer of soil is discovered in the lower part of the embankment as shown in Figure 6.61.

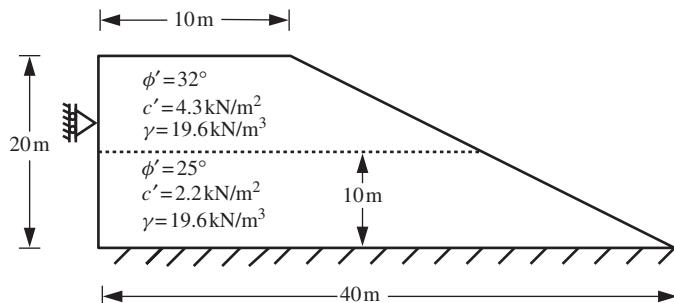


Figure 6.61

Answer:  $F \approx 1.06$

7. Use Program 6.4 to estimate the factor of safety of the slope shown in Figure 6.62 with  $c_{u2} = 4, 7.5$  and  $10 \text{ kN/m}^2$ .

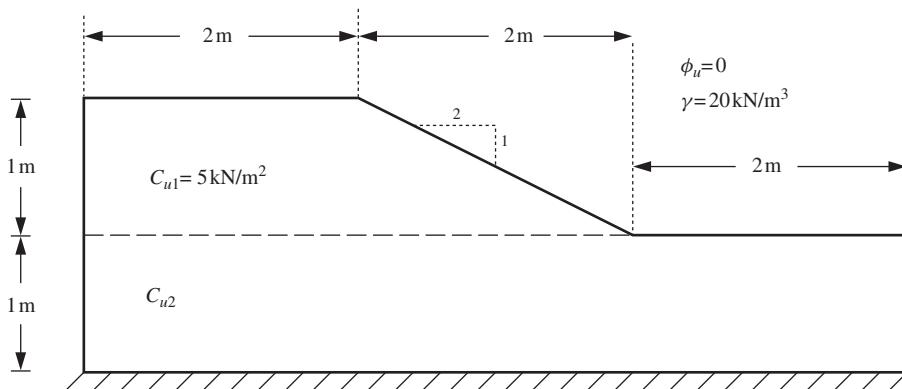


Figure 6.62

Answer:  $F \approx 1.22, 2.05, 2.09$

8. Modify the data shown in Figure 6.19 to estimate the factor of safety of the slope if the undrained shear strength ( $\phi_u = 0$ ) increases linearly from  $20 \text{ kN/m}^2$  at the top surface to  $50 \text{ kN/m}^2$  at 15 m depth.

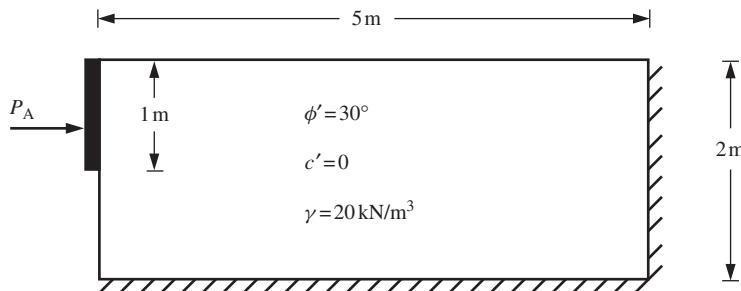


Figure 6.63

Answer:  $FS \approx 1.29$

9. Use Program 6.5 to estimate the active force exerted by the soil on the wall shown in Figure 6.63.

Answer:  $F \approx 3.3 \text{ kN/m}$

10. Use Program 6.11 to repeat the excavation example described in Figure 6.45 using the construction sequence of Cases D and E in Figure 6.44.

Answer: Case D fails after first excavation of elements 9 and 10. Case E fails after fourth excavation of element 14

11. Use Program 6.12 or 6.13 to investigate the influence of the third dimension  $d_1$  on the factor of safety of the cohesive slope shown in Figure 6.64. Gradually increase the depth  $d_1$  of the mesh and use the symmetry boundary condition `ifix=2`.

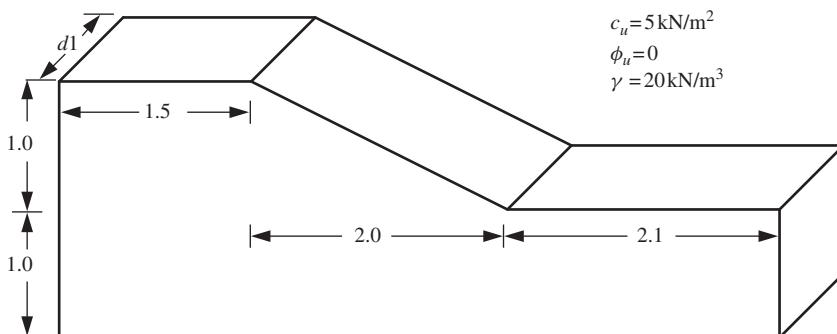


Figure 6.64

	$d_1$	1.4	2.1	2.8	3.5	4.2	plane strain
Answer:	$F$	1.97	1.75	1.66	1.59	1.56	1.45

## References

- Belytschko T, Liu WK and Moran B 2000 *Nonlinear Finite Elements for Continua and Structures*. John Wiley & Sons, Chichester.
- Bishop AW and Morgenstern NR 1960 Stability coefficients for earth slopes. *Géotechnique* **10**, 129–150.
- Cormeau IC 1975 Numerical stability in quasi-static elasto-viscoplasticity. *Int J Numer Methods Eng* **9**(1), 109–127.
- Crisfield MA 1997 *Nonlinear Finite Element Analysis of Solids and Structures. Advanced Topics, Vol 2*. John Wiley & Sons, Chichester.
- Duncan JM and Chang CY 1970 Nonlinear analysis of stress and strain in soils. *J Soil Mech Found Div, ASCE* **96**(SM5), 1629–1653.
- Griffiths DV 1980 *Finite element analyses of walls, footings and slopes*. PhD thesis, Department of Engineering, University of Manchester.
- Griffiths DV 1982 Computation of bearing capacity factors using finite elements. *Géotechnique* **32**(3), 195–202.
- Griffiths DV 1985 The effect of pore fluid compressibility on failure loads in elasto-plastic soils. *Int J Numer Anal Methods Geomech* **9**, 253–259.
- Griffiths DV and Lane PA 1999 Slope stability analysis by finite elements. *Géotechnique* **49**(3), 387–403.
- Griffiths DV and Marquez RM 2007 Three-dimensional slope stability analysis by elasto-plastic finite elements. *Géotechnique* **57**(6), 537–546.
- Griffiths DV and Mustoe GGW 1995 Selective reduced integration of the four node plane element in closed-form. *J Eng Mech, ASCE* **121**(6), 725–729.
- Griffiths DV and Willson SM 1986 An explicit form of the plastic matrix for a Mohr–Coulomb material. *Comm Appl Numer Methods* **2**, 523–529.
- Hill R 1950 *The Mathematical Theory of Plasticity*. Oxford University Press, Oxford.
- Huang J and Griffiths DV 2008 Observations on return mapping algorithms for piecewise linear yield criteria. *Int J Geomech* **8**(4), 253–265.
- Huang J and Griffiths DV 2009 Return mapping algorithms and stress predictors for failure analysis in geomechanics. *J Eng Mech* **135**(4), 276–284.
- Hughes TJR 1987 *The Finite Element Method*. Prentice-Hall, Englewood Cliffs, NJ.
- Krieg RD and Krieg OB 1977 Accuracies of numerical solution methods for elastic–perfectly plastic models. *ASME J Press Vessel Technol* **99**, 510–515.
- Martin CM 2004 User guide for ABC–analysis of bearing capacity. Technical Report OUEL 2261/03, University of Oxford, Department of Engineering Science.
- Molenkamp F 1987 Kinematic model for alternating loading ALTERNAT. Technical Report CO-218 598, Delft Geotechnics.
- Nayak GC and Zienkiewicz OC 1972 Elasto/plastic stress analysis. A generalisation for various constitutive relationships including strain softening. *Int J Numer Methods Eng* **5**, 113–135.
- Naylor DJ 1974 Stresses in nearly incompressible materials by finite elements with application to the calculation of excess pore pressure. *Int J Numer Methods Eng* **8**, 443–460.
- Ortiz M and Popov EP 1985 Accuracy and stability integration algorithms for elastoplastic constitutive relations. *Int J Numer Methods Eng* **21**, 1561–1576.
- Rice JR and Tracey DM 1973 Computational fracture mechanics. In *Proc Symp Num Meths Struct Mech* (ed. Fenves S). Academic Press, New York.
- Simo JC and Taylor RL 1985 Consistent tangent operators for rate independent elastoplasticity. *Comput Methods Appl Mech Eng* **48**, 101–108.
- Smith IM 1997 Computation of large scale viscoplastic flows of frictional geotechnical materials In *Dynamics of Complex Fluids* (eds Adams MJ et al.). Imperial College Press, London.
- Smith IM and Ho DKH 1992 Influence of construction technique on performance of braced excavation in marine clay. *Int J Numer Anal Methods Geomech* **16**, 845.
- Smith IM and Hobbs R 1974 Finite element analysis of centrifuged and built-up slopes. *Géotechnique* **24**(4), 531–559.
- Taylor DW 1937 Stability of earth slopes. *J Boston Soc Civ Eng* **24**, 197–246.
- Yamada Y, Yoshimura N and Sakurai T 1968 Plastic stress–strain matrix and its application for the solution of elastic plastic problems by the finite element method. *J Mech Sci* **10**, 343–354.
- Zienkiewicz OC and Cormeau IC 1974 Visco-plasticity–plasticity and creep in elastic solids—a unified numerical solution approach. *Int J Numer Methods Eng* **8**(4), 821–845.
- Zienkiewicz OC, Valliappan S and King IP 1969 Elasto-plastic solutions of engineering problems, ‘initial stress’ finite element approach. *Int J Numer Methods Eng* **1**, 75–100.
- Zienkiewicz OC, Humpheson C and Lewis RW 1975 Associated and non-associated viscoplasticity and plasticity in soil mechanics. *Géotechnique* **25**(4), 671–689.
- Zienkiewicz OC, Taylor RL and Zhu JZ 2005 *The Finite Element Method*, Vol. 1, 6th edn. McGraw-Hill, London.



# 7

# Steady State Flow

## 7.1 Introduction

The five programs presented in this chapter solve steady-state problems governed by Laplace's equation (2.125). Typical examples of this type of problem include steady seepage through soils and steady heat flow through a conductor. Examples are presented of planar (confined and unconfined), axisymmetric and three-dimensional flow. Unlike the problems solved in Chapters 5 and 6, which gave vector fields of displacements, the dependent variable in these problems is a scalar, generically called the 'potential' which may represent, for example, the total head in a seepage problem or the temperature in a heat flow analysis. Each node therefore has only one degree of freedom associated with it.

Systems that are governed by Laplace's equation require boundary conditions to be prescribed at all points around a closed domain. These boundary conditions commonly take the form of fixed values of the potential or its first derivative normal to the boundary. The problem amounts to finding the values of the potential at points within the closed domain.

Being 'elliptic' in character, solution of Laplace's equation quite closely resembles the solution of equilibrium equations (2.57) in solid elasticity. Both methods ultimately require the solution of a set of linear simultaneous equations. The element conductivity matrix (analogous to the stiffness matrix in elasticity) can be formed numerically, as described by (3.69) or 'analytically', as discussed in Section 3.2.2. Either way, the element matrices can be assembled into a global conductivity matrix which, like its global elastic counterpart, is symmetrical, banded and usually stored as a skyline. Alternatively, element-by-element iterative strategies can be used. Taking the analogy with Chapter 5 one stage further, 'displacements' now become total heads and 'loads' become net nodal inflow.

Program 7.1 describes the solution of Laplace's equation over a set of 1D elements that can each have different lengths, areas, and permeabilities. The elements can be attached end to end, or in any desired 'network' of connections. Program 7.2 describes the solution of Laplace's equation over a plane or axisymmetric 2D domain. Program 7.3 describes the non-linear problem of free surface flow, in which the mesh is allowed to deform iteratively until it assumes the shape of the free surface at convergence. Program 7.4 is a general program for the solution of Laplace's equation over two- or three-dimensional domains. The final Program 7.5 describes an element-by-element version of Program 7.4, avoiding the need for global matrix assembly. A parallel version of this program is also described in Chapter 12.

As the problems in this chapter involve scalar fields with just one unknown at each node, the programming is simplified in that `nod` is always equal to `ndof`, so the latter is not required. A further change from the solid mechanics applications of the preceding chapters is that the ‘zero freedoms’ data previously introduced through the `nf` array has been removed. In Chapter 7, all fixed boundary conditions, whether equal to zero or not, are applied through the `fixed_freedom` data. Since all nodal freedoms are therefore retained in the assembly and analysis, `nn` is always equal to `neq`. In the interests of consistency with other programs in the book, however, `neq` has been retained.

## Program 7.1 One-dimensional analysis of steady seepage using 2-node line elements

```

PROGRAM p71
!-----
! Program 7.1 One dimensional analysis of steady seepage using
!           2-node "rod" elements.
!-----
USE main; IMPLICIT NONE; INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,nel,neq,nlen,nod=2,nn,
nprops=1,np_types
REAL(iwp)::penalty=1.0e20_iwp,zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),kdiag(:),g_num(:,:,),node(:),num(:)
REAL(iwp),ALLOCATABLE::disps(:),ell(:),kp(:,:,),kv(:,),kvh(:,),loads(:),
prop(:,:,),value(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nel,nn,np_types; neq=nn
ALLOCATE(ell(nel),num(nod),prop(nprops,np_types),etype(nel),
kp(nod,nod),g_num(nod,nel),kdiag(neq),loads(0:neq),disps(0:neq))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)ell; READ(10,*)g_num; kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nel
    num=g_num(:,:,iel); CALL fkdiag(kdiag,num)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
WRITE(11,'(2(A,I5))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq))); kv=zero
!-----global conductivity matrix assembly-----
elements_2: DO iel=1,nel
    CALL rod_km(kp,prop(1,etype(iel)),ell(iel))
    num=g_num(:,:,iel); CALL fsparv(kv,kp,num,kdiag)
END DO elements_2; kvh=kv
!-----specify boundary values-----
loads=zeros; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    kv(kdiag(node))=kv(kdiag(node))+penalty
    loads(node)=kv(kdiag(node))*value
END IF

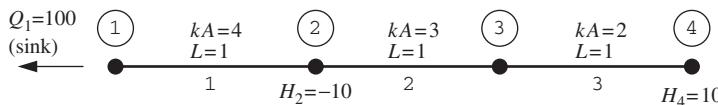
```

```

!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
!-----retrieve nodal net flow rates-----
CALL linmul_sky(kvh,loads,disps,kdiag)
WRITE(11,'(/A)')" Node Total Head Flow rate"; disps(0)=zero
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
WRITE(11,'(/A)')" Inflow Outflow"
WRITE(11,'(5X,2E12.4)')
SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero)
STOP
END PROGRAM p71
&

```

Figure 7.1 shows a string of three elements attached end to end. Each element has the same length, but different permeability properties. In this context, the property applied to each element is  $kA$  (analogous to  $EA$  in Program 4.1), namely the product of the permeability and the cross-sectional area of each element. A fixed steady outflow or ‘sink’ of  $-100.0$  (negative sign denotes outflow) is applied at node 1, and the total head is fixed to  $-10.0$  and  $10.0$  at nodes 2 and 4, respectively. The data involves reading the number of elements `nels`, the number of nodes `nn`, and the number of property types `np_types`. In this case there are three property types, one for each element, so with  $np\_types > 1$  the `etype` data is read next, indicating that element 1 has a  $kA$  of 4.0, element 2 has a  $kA$  of 3.0 and so on. The element lengths `ell` are read, followed by the node numbers of each element `g_num`. In the case of a string of elements such as this, the `g_num` data is quite predictable. Finally, the fixed source/sink values and fixed total



```

n els   nn   np_types
3      4      3

prop(ka)
4.0   3.0   2.0

etype
1 2 3

ell
1.0   1.0   1.0

g_num
1 2   2 3   3 4

loaded_nodes,(k,loads(k),i=1,loaded_nodes)
1
1   -100.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
2
2   -10.0   4   10.0

```

**Figure 7.1** Mesh and data for first Program 7.1 example

head values are read through the `loaded_nodes` and `fixed_freedoms` data. The output shown in Figure 7.2 indicates the total head at nodes 1 and 3 to be  $-35.00$  and  $-2.00$ , respectively, and the net flow rates at nodes 2 and 4 to be inflows of  $76.0$  and  $24.0$ , respectively. The final line of output confirms the continuity condition that the total flow in, is the same as the total flow out.

The second example shown in Figure 7.3 is of a pipe network involving six elements and five nodes. There are three different property groups spread across the elements,

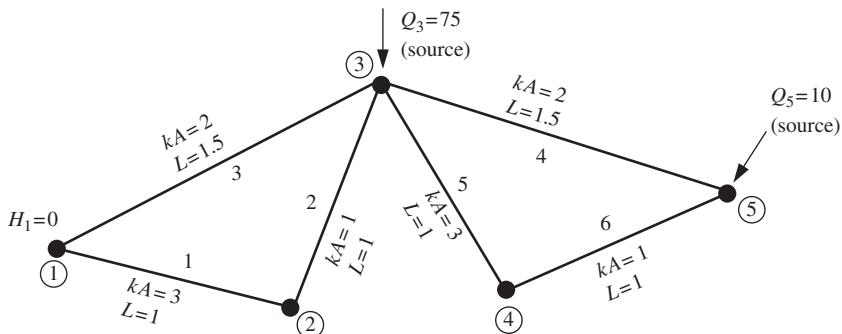
There are 4 equations and the skyline storage is 7

Node	Total Head	Flow rate
1	-0.3500E+02	-0.1000E+03
2	-0.1000E+02	0.7600E+02
3	-0.2000E+01	0.3553E-14
4	0.1000E+02	0.2400E+02

Inflow	Outflow
0.1000E+03	-0.1000E+03

**Figure 7.2** Results from first Program 7.1 example



```

nels    nn    np_types
6       5      3

prop(ka)
3.0   2.0   1.0

etype
1   3   2   2   1   3

ell
1.0   1.0   1.5   1.5   1.0   1.0

g_num
1 2   2 3   1 3   3 5   3 4   4 5

loaded_nodes,(k,loads(k),i=1,loaded_nodes)
2
3 75.0   5   10.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
1
1   0.0

```

**Figure 7.3** Mesh and data for second Program 7.1 example

```

There are      5 equations and the skyline storage is   11

Node Total Head  Flow rate
 1  0.8500E-18 -0.8500E+02
 2  0.1020E+02  0.0000E+00
 3  0.4080E+02  0.7500E+02
 4  0.4200E+02 -0.2842E-13
 5  0.4560E+02  0.1000E+02

Inflow          Outflow
 0.8500E+02 -0.8500E+02

```

**Figure 7.4** Results from second Program 7.1 example

which do not all have the same lengths. The `g_num` data gives the connectivity of the network. The boundary conditions include sources of 75.0 and 10.0 at nodes 3 and 5, respectively, and a total head at node 1 equal to zero. The output shown in Figure 7.4 indicates an outflow of 85.0 at node 1, and total heads at nodes 2, 3, 4 and 5 of 10.2, 40.8, 42.0 and 45.6, respectively.

It should be noted that in problems of this type, all nodal boundary conditions are fixed to either a net flow rate or a total head. If the data prescribes both the net flow rate and the total head at a particular node, the total head takes priority as the dependent variable. The default net flow rate at all nodes is set initially to zero, so only non-zero values need to be input as data.

## Program 7.2 Plane or axisymmetric analysis of steady seepage using 4-node rectangular quadrilaterals. Mesh numbered in $x(r)$ - or $y(z)$ -direction

```

PROGRAM p72
!-----
! Program 7.2 Plane or axisymmetric analysis of steady seepage using
!           4-node rectangular quadrilaterals. Mesh numbered
!           in x(r)- or y(z)- direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,nci,ndim=2,nels,neq,nip=4,  &
nlen,nod=4,nn,np_types,nxe,nye
REAL(iwp)::det,one=1.0_iwp,penalty=1.0e20_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,dir,element='quadrilateral',type_2d
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g_num(:,:),kdiag(:),node(:),num(:)
REAL(iwp),ALLOCATABLE::coord(:,:),der(:,:),deriv(:,:),disps(:),fun(:),  &
gc(:,),g_coord(:,:),jac(:,:),kay(:,:),kp(:, :,),kv(:, ),kvh(:, ),loads(:, ) ,&
points(:,:),prop(:,:),value(:, ),weights(:, ),x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,dir,nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),  &
jac(ndim,ndim),weights(nip),der(ndim,nod),deriv(ndim,nod),  &
kp(nod,nod),num(nod),g_num(nod,nels),kay(ndim,ndim),etype(nels),  &
x_coords(nxe+1),y_coords(nye+1),prop(ndim,np_types),gc(ndim),fun(nod),  &
kdiag(neq),loads(0:neq),disps(0:neq))

```

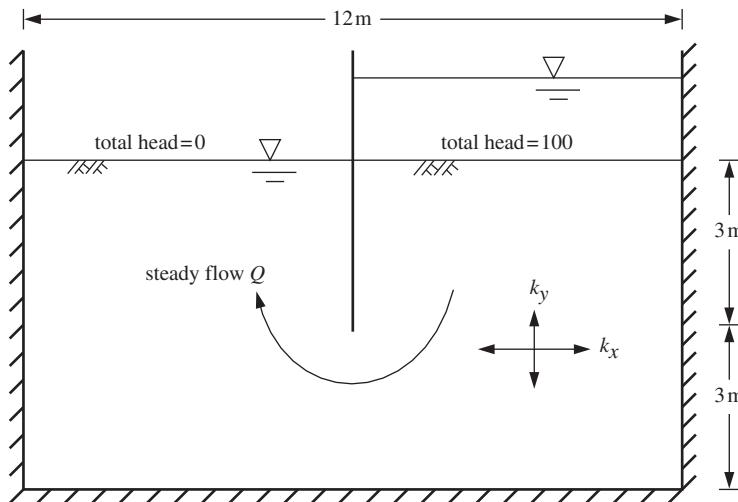
```

READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords; kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
    CALL fkdiag(kdiag,num)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq))); WRITE(11,'(2(A,I5))')      &
    "There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); kv=zero; gc=one
!-----global conductivity matrix assembly-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kp=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); IF(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
        kp=kp+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
    END DO gauss_pts_1; CALL fsparv(kv,kp,num,kdiag)
END DO elements_2; kvh=kv
!-----specify boundary values-----
loads=zero; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    kv(kdiag(node))=kv(kdiag(node))+penalty
    loads(node)=kv(kdiag(node))*value
END IF
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
!-----retrieve nodal net flow rates-----
CALL linmul_sky(kvh,loads,disps,kdiag)
WRITE(11,'(/A)')" Node Total Head Flow rate"
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
disps(0)=zero; WRITE(11,'(/A)')" Inflow Outflow"
WRITE(11,'(5X,2E12.4)')      &
    SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero); READ(10,*)nci
IF(nod==4)CALL contour(loads,g_coord,g_num,nci,argv,nlen,13)
STOP
END PROGRAM p72

```

This program is for the analysis of 2D steady seepage problems under plane or axisymmetric conditions, and is analogous to Program 5.1 in Chapter 5. In order to simplify the data, however, the examples presented here use 4-node rectangular elements only (element='quadrilateral' and nod=4). The program includes graphics subroutines `mesh` and `contour` which generate PostScript files containing, respectively, images of the finite element mesh (held in `*.msh`), and a contour map of the dependent variable (held in `*.con`). Contouring will currently only work for meshes made up of 4-node quadrilateral elements.

The first example in Figure 7.5 shows a typical problem of steady seepage beneath an impermeable sheet pile wall. The total head loss across the wall has been normalised to



**Figure 7.5** Steady flow under a single sheet pile

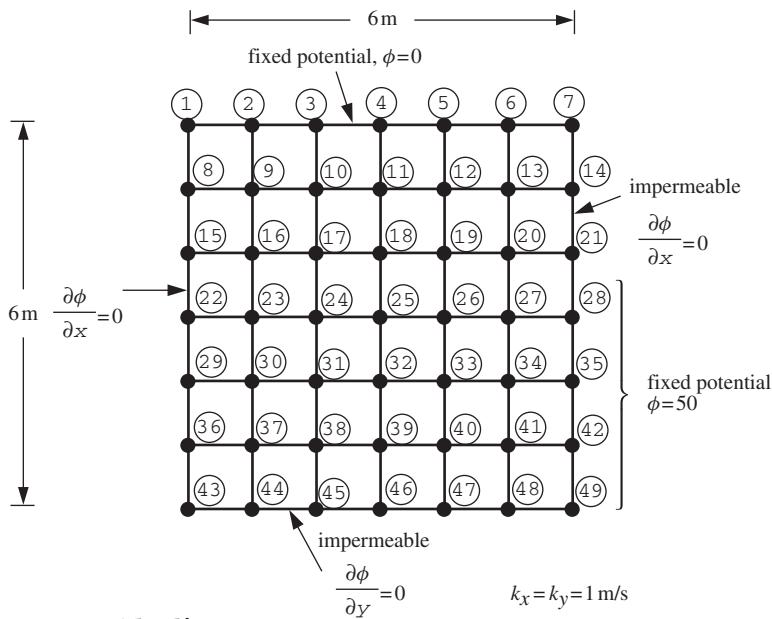
100 units, but due to symmetry only half the problem needs to be analysed, with a total head loss of 50 units. Figure 7.6 shows the mesh and data that will be used to analyse the problem.

The first line of data reads `type_2d` and `dir`, indicating that a plane analysis is to be performed with element and node numbering in the  $x$ -direction. The second line indicates that the rectangular mesh consists of six columns (`nxe`) and six rows (`nye`) of elements, and that there is only one property type (`np_types`) in this homogeneous example. The third line reads the  $x$ - and  $y$ -direction permeabilities,  $k_x$  and  $k_y$  into the property array `prop`, and since there is only one property type in this problem, the `etype` data is not required. The fourth and fifth lines give, respectively, the  $x$ - (`x_coords`) and  $y$ -coordinates (`y_coords`) of the lines that make up the mesh. The zero `loaded_nodes` on the sixth lines indicates that no internal sources or sinks are applied in this case. The seventh to tenth lines indicate that there are 11 fixed total head values (`fixed Freedoms`) at the up- and downstream boundaries of the mesh. The last line of data indicates that after the total head values have been computed, a contour map will be produced with 10 (`nci`) contour intervals [or equipotential drops  $n_d$ , see equation (7.2)].

The fixed potential boundary conditions (equal to either zero or 50) are fixed through the data as described above. All other boundaries are ‘no-flow’ or impermeable, so a boundary condition of  $\partial\phi/\partial n = 0$  is required, which is obtained by default at the boundaries of the mesh by taking no further action.

The program assumes a rectangular mesh made up of rectangular elements, with nodal coordinates and connectivity generated by the library subroutine `geom_rect`.

After scanning the elements to determine the storage requirements, the program uses numerical integration to form the element conductivity matrices `kc`, which are then assembled into a global conductivity matrix `kv`. The sequence of operations described by the `elements_2` loop bears a striking similarity to the integration of an element stiffness matrix used, for example, in Program 5.1. Program 7.2 is actually simpler, because the derivative array `deriv` is used directly in the products described by (3.69).



```

type_2d dir
'plane' 'x'

nxe nye np_types
6 6 1

prop(kx,ky)
1.0 1.0

etype(not needed)

x_coords, y_coords
0.0 1.0 2.0 3.0 4.0 5.0 6.0
0.0 -1.0 -2.0 -3.0 -4.0 -5.0 -6.0

loaded_nodes
0

fixed_freedoms, (node(i),value(i),i=1,fixed_freedoms)
11
1 0.0 2 0.0 3 0.0 4 0.0
5 0.0 6 0.0 7 0.0
28 50.0 35 50.0 42 50.0 49 50.0

nci
10

```

**Figure 7.6** Mesh and data for first Program 7.2 example

Following solution of the ‘equilibrium’ equations which is performed by library subroutines sparin and spabac, the nodal potentials are held in the vector loads and printed. In order to retrieve the nodal flow rates disps, the matrix kvh, which is a copy of the global conductivity matrix kv, is multiplied by the nodal potentials loads by library subroutine linmul\_sky. Examination of disps reveals that the majority of net flow rates corresponding to internal nodes are zero, the only non-zero values occurring at

There are 49 equations and the skyline storage is 385

Node	Total Head	Flow rate
1	0.2926E-19	-0.2926E+01
2	0.6063E-19	-0.6063E+01
3	0.6708E-19	-0.6708E+01
4	0.7810E-19	-0.7810E+01
5	0.9184E-19	-0.9184E+01
6	0.1042E-18	-0.1042E+02
7	0.5453E-19	-0.5453E+01
8	0.5716E+01	-0.8882E-15
9	0.5921E+01	-0.1776E-14
10	0.6553E+01	0.0000E+00
.	.	.
.	.	.
40	0.3378E+02	0.1066E-13
41	0.4130E+02	0.0000E+00
42	0.5000E+02	0.9201E+01
43	0.2196E+02	-0.2220E-14
44	0.2272E+02	-0.1776E-14
45	0.2502E+02	0.7105E-14
46	0.2897E+02	0.1510E-13
47	0.3464E+02	0.0000E+00
48	0.4185E+02	-0.1776E-13
49	0.5000E+02	0.4260E+01
Inflow	Outflow	
0.4857E+02	-0.4857E+02	

**Figure 7.7** Results from first Program 7.2 example

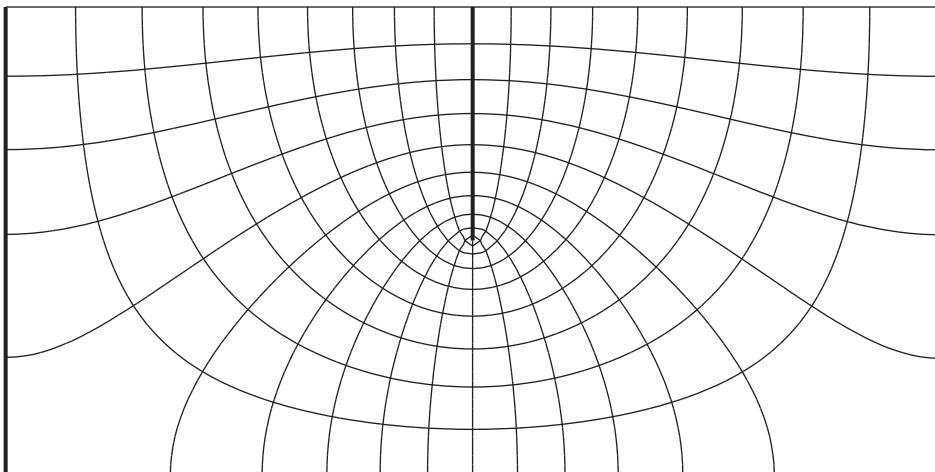
the boundary nodes that had their total head values fixed. If we had chosen to include an internal source or sink as data using `loaded_nodes`, this would have appeared at the appropriate node in the `disps` vector.

Finally, the net inflow and outflow through the system is computed by summing, respectively, the positive and negative terms in `disps`. The output from Program 7.2 is shown in Figure 7.7. As expected, the inflow and outflow values are identical and give a steady-state flow rate of 48.6. The method of fragments for this constrained seepage problem (e.g., Griffiths, 1984) would predict a flow rate of around 47. The theoretical solution for a sheet pile wall embedded to half the depth of a stratum of similar soil in a domain which extends to infinity laterally would be exactly 50.0.

A good way to visualise the results of a seepage analysis such as this is to draw a contour map of the nodal potentials. Figure 7.8 shows a contour map of the total heads and the stream functions that would be computed using a rather more refined mesh ( $50 \times 50$  elements) than that shown in Figure 7.6. Both sides of the wall are shown for clarity, although only half the problem was actually analysed. The stream function problem has not been solved in this example, however, it could easily be included by solving the ‘inverse’ problem given by

$$\frac{1}{k_y} \frac{\partial^2 \psi}{\partial x^2} + \frac{1}{k_x} \frac{\partial^2 \psi}{\partial y^2} = 0 \quad (7.1)$$

where  $\psi$  is the stream function. The boundary conditions for the stream problem must now be ‘inverted’, thus those boundaries that had fixed values in the potential problem such as the up- and downstream boundaries, have  $\partial\psi/\partial n = 0$  boundary conditions in the



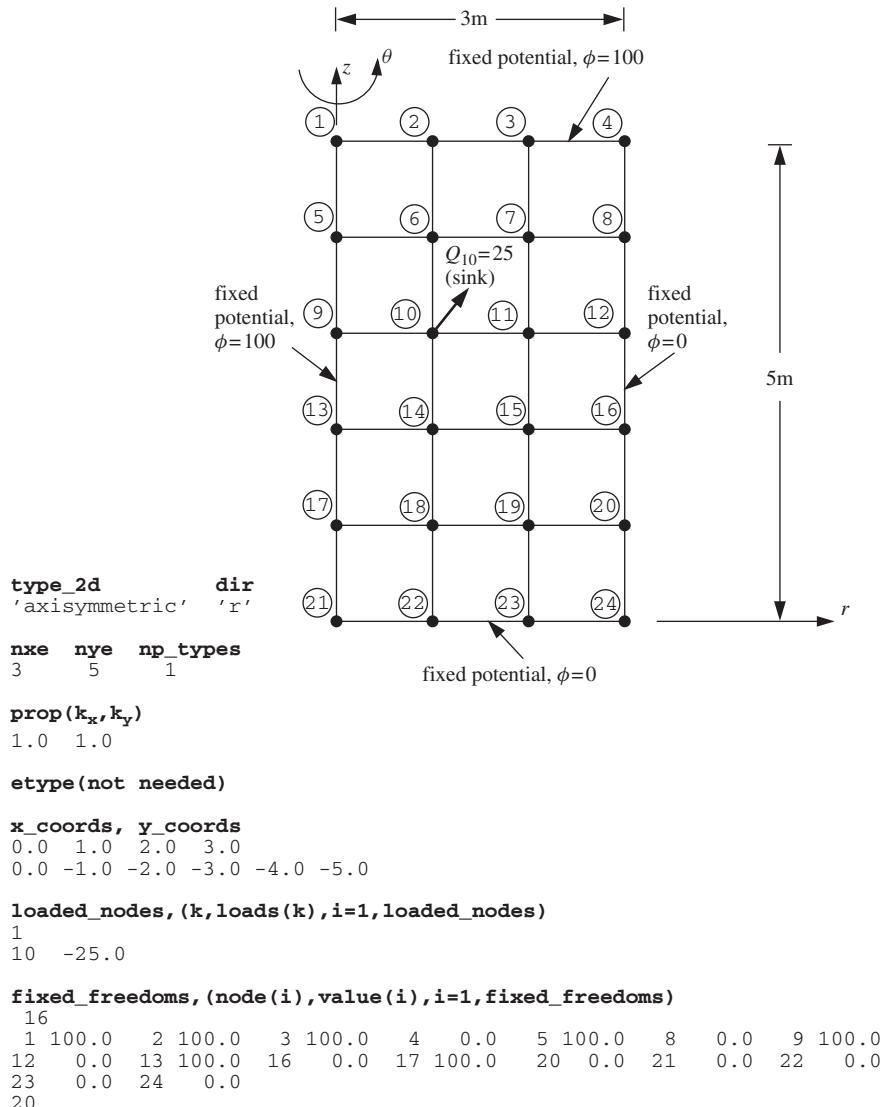
**Figure 7.8** Flow net of seepage beneath a single sheet pile from first Program 7.2 example

stream problem, and boundaries that had  $\partial\phi/\partial n = 0$  conditions in the potential problem, such as at impermeable boundaries, are given fixed values of the stream function. In order to choose a contour interval which satisfies the usual flow-net rules involving ‘square’ regions, it is suggested that when solving the stream problem, the uppermost streamline (in this case the wall) is fixed equal to the flow rate ( $\psi = 48.6$ ), and the lowest streamline (the impermeable boundary) is fixed to zero ( $\psi = 0$ ). The required number of flow channels  $n_f$  for the stream contour map can then be computed from

$$n_f = \frac{Q}{k} \frac{n_d}{H} \quad (7.2)$$

where  $Q$  is the flow rate computed from the potential problem,  $k$  is the (isotropic) permeability of the soil,  $n_d$  is the number of equipotential drops chosen for the total head contour plot and  $H$  is the total head loss between the up- and downstream boundaries. Finally,  $n_f$  should be rounded to the nearest whole number and input to the stream function analysis as the number of stream contour intervals nci.

A second example of the use of Program 7.2 is shown in Figure 7.9, and represents a radial plane of a cylinder of porous material. The model subtends one radian at the axis of rotational symmetry. The boundary conditions consist of a fixed total head of 100 units on the top of the cylinder and on the central axis ( $r = 0$ ). The outer surface of the cylinder and the bottom surface are fixed to zero. The data are very similar to those of the previous example, but with type\_2d set to ‘axisymmetric’ and dir set to ‘r’ because the node and element numbering is now in the radial direction. Although axisymmetry adds an extra order of  $r$  to the terms to be integrated, the conductivity matrix of rectangular 4-node elements is still exactly integrated with nip=4. When performing axisymmetric analysis with non-rectangular quadrilateral elements, however, higher orders of numerical integration should be investigated, and slightly different results can be expected as nip



**Figure 7.9** Mesh and data for second Program 7.2 example

is increased. Customised quadrature rules for axisymmetric analysis are available (see, e.g., Griffiths, 1991).

In this example, a steady point sink of  $-25.0 \text{ m}^3/\text{s/radian}$  is applied to node 10. The computed results are shown in Figure 7.10. In addition to the usual flow rates recorded at the boundary nodes, the fluid removed from the system at node 10 also appears in the ‘Flow rate’ column as  $-25.0$ . The net inflow (outflow) from the entire system is computed to be  $362.7 \text{ m}^3/\text{s/radian}$ .

```

There are    24 equations and the skyline storage is 122

Node Total Head   Flow rate
 1  0.1000E+03  0.6068E+01
 2  0.1000E+03  0.4558E+02
 3  0.1000E+03  0.1878E+03
 4  0.6925E-18 -0.6925E+02
 5  0.1000E+03  0.1726E+02
 6  0.6359E+02  0.3553E-14
 7  0.3310E+02 -0.4086E-13
 8  0.1255E-17 -0.1255E+03
 9  0.1000E+03  0.2863E+02
10  0.3283E+02 -0.2500E+02
.
.
.
19  0.6183E+01  0.7105E-14
20  0.1421E-18 -0.1421E+02
21  0.3278E-19 -0.3278E+01
22  0.2631E-18 -0.2631E+02
23  0.1396E-18 -0.1396E+02
24  0.5152E-19 -0.5152E+01

Inflow          Outflow
0.3627E+03 -0.3627E+03

```

**Figure 7.10** Results from second Program 7.2 example

### Program 7.3 Analysis of plane free surface flow using 4-node quadrilaterals. 'Analytical' form of element conductivity matrix

```

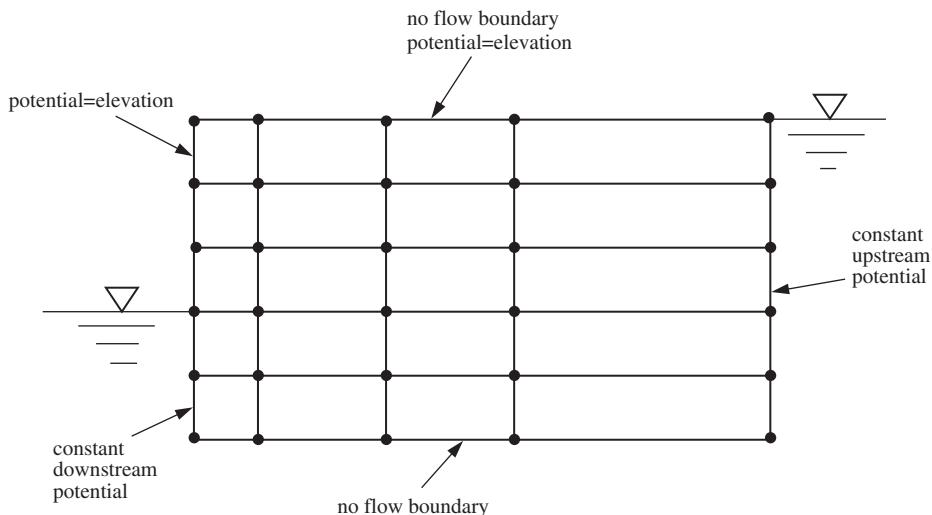
PROGRAM p73
!-----
! Program 7.3 Analysis of plane free-surface flow using 4-node
!           quadrilaterals. "Analytical" form of element conductivity
!           matrix.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_down,fixed_seep,fixed_up,i,iel,iter,k,limit,ncl,ndim=2,  &
  nels,neq,nlen,nod=4,nn,nxe,nye,np_types; CHARACTER(LEN=15)::argv
REAL(iwp)::d180=180.0_iwp,hdown,hup,initial_height,one=1.0_iwp,      &
  penalty=1.e20_iwp,tol,zero=0.0_iwp; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g_num(:,:),kdiag(:),node_down(:),        &
  node_seep(:),node_up(:),num(:)
REAL(iwp),ALLOCATABLE::angs(:,bottom_width(:),coord(:,:),disps(:),      &
  g_coord(:,:),kay(:,:),kp(:,:),kv(:),kvh(:),loads(:),oldpot(:),    &
  prop(:,:),surf(:,top_width(:))
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,tol,limit,np_types; nels=nxe*nye
nn=(nxe+1)*(nels/nxe+1); neq=nn
ALLOCATE(g_coord(ndim,nn),coord(nod,ndim),bottom_width(nxe+1),        &
  top_width(nxe+1),surf(nxe+1),angs(nxe+1),kp(nod,nod),num(nod),       &
  g_num(nod,nels),prop(ndim,np_types),kdiag(neq),kay(ndim,ndim),       &
  etype(nels),loads(0:neq),disps(0:neq),oldpot(0:neq))

```

```

READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)bottom_width; READ(10,*)top_width; READ(10,*)initial_height
surf=initial_height
angs=ATAN(surf/(top_width-bottom_width))*d180acos(-one)
READ(10,*)hup,fixed_up; ALLOCATE(node_up(fixed_up)); READ(10,*)node_up
READ(10,*)hdown,fixed_down; ALLOCATE(node_down(fixed_down))
READ(10,*)node_down; fixed_seep=nels/nxe-fixed_down
ALLOCATE(node_seep(fixed_seep))
DO i=1,fixed_seep; node_seep(i)=i*(nxe+1)+1; END DO; kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_freesurf(iel,nxe,fixed_seep,fixed_down,
        hdown,bottom_width,angs,surf,coord,num); CALL fkdiag(kdiag,num)
    &
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq)))
WRITE(11,'(2(A,I5))')
&
"There are ",neq," equations and the skyline storage is ",kdiag(neq)
!-----global conductivity matrix assembly-----
oldpot=zero; iters=0
its: DO
    iters=iters+1; kv=zero
    elements_2: DO iel=1,nels
        kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
        CALL geom_freesurf(iel,nxe,fixed_seep,fixed_down,
            hdown,bottom_width,angs,surf,coord,num)
        g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
        CALL seep4(kp,coord,kay); CALL fsparv(kv,kp,num,kdiag)
    END DO elements_2; kvh=kv
!-----specify boundary values-----
loads=zero; kv(kdiag(node_up))=kv(kdiag(node_up))+penalty
loads(node_up)=kv(kdiag(node_up))*hup
kv(kdiag(node_down))=kv(kdiag(node_down))+penalty
loads(node_down)=kv(kdiag(node_down))*hdown
kv(kdiag(node_seep))=kv(kdiag(node_seep))+penalty
DO i=1,fixed_seep
    loads(node_seep(i))=kv(kdiag(node_seep(i)))*
        (hdown+(surf(1)-hdown)*(fixed_seep+1-i)/(fixed_seep+1))
    &
END DO
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
surf(1:nxe)=loads(1:nxe)
!-----check convergence-----
CALL checon(loads,oldpot,tol,converged)
IF(converged.OR.iters==limit)EXIT
END DO its; CALL linmul_sky(kvh,loads,disps,kdiag)
WRITE(11,'(/A)')" Node Total Head Flow rate"
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
disps(0)=zero; WRITE(11,'(/A)')" Inflow Outflow"
WRITE(11,'(5X,2E12.4)')
    SUM(disps,MASK=disps<zero),SUM(disps,MASK=disps>zero)
WRITE(11,'(/A,I3,A)')" Converged in",iters," iterations"
CALL mesh(g_coord,g_num,argv,nlen,12)
READ(10,*)nci; CALL contour(loads,g_coord,g_num,nci,argv,nlen,13)
STOP
END PROGRAM p73

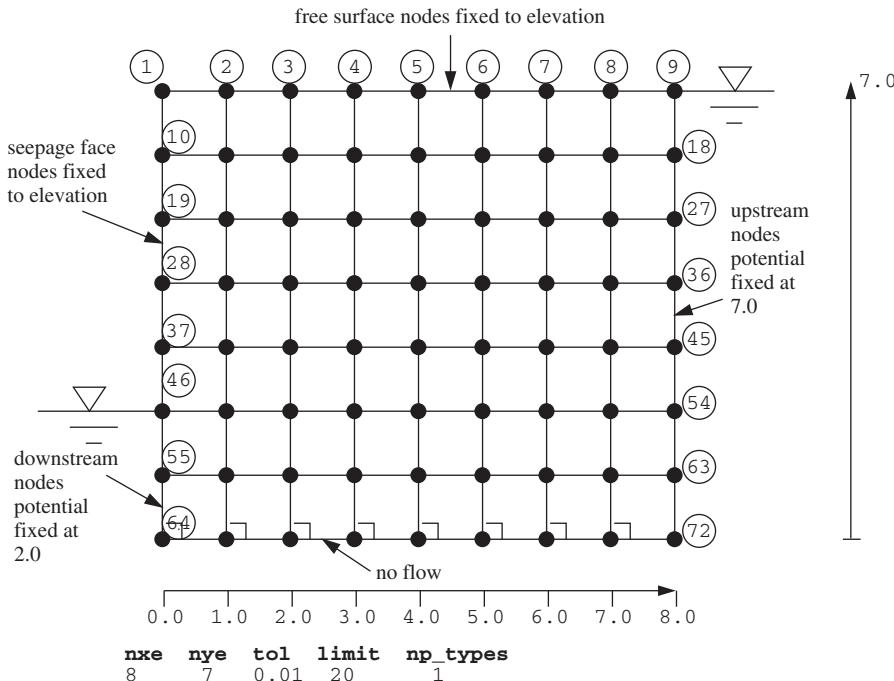
```



**Figure 7.11** Boundary conditions for free surface flow

In this program we consider a boundary condition frequently met in geomechanics in relation to flow of water through dams. Free surface problems involve an upper boundary, the location of which is not known a priori so an iterative procedure is required to find it. This iteration can be done in several ways; for example, a fixed mesh can be used and nodes separated into ‘active’ and ‘inactive’ ones depending upon whether fluid exists at that point. An alternative strategy is to use the present program, whereby the mesh is deformed so that its upper surface ultimately coincides with the free surface. A summary of the boundary conditions is given in Figure 7.11.

The analysis starts by assuming an initial position for the free surface. Solution of Laplace’s equation gives values of the total head along the free surface nodes which will not in general equal the elevation of the upper surface of the mesh. The elevations of the nodes along the upper surface are therefore adjusted to equal the total head values just calculated at those locations. In order to avoid distorted elements, the library geometry subroutine `geom_freesurf` ensures that the nodes beneath the top surface are evenly distributed. The geometry subroutine is designed for solving free surface problems with initially trapezoidal meshes and counts nodes and elements in the  $x$ -direction. The analysis is then repeated with the new mesh. Since many of the coordinates have changed, the conductivity matrices of all the elements must be recomputed and assembled into the global system. In order to avoid the need for numerical integration of the element conductivity matrices at each iteration, library subroutine `seep4` computes the element conductivity matrices `kc` ‘analytically’ (see Section 3.2.2). The assembly is made into a global conductivity matrix `kv` stored as a skyline in the usual way. Solution of the modified problem leads to another set of nodal total head values and further updating of the mesh nodal coordinates. This process is repeated until the change in computed total head values from one iteration to the next is less than a tolerance value `tol`. The convergence check is performed by library subroutine `checn`, which outputs the logical



```

nxe nye tol limit np_types
8    7    0.01 20      1

prop(kx,ky)
0.001 0.001

etype(not needed)

bottom_width, top_width
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0

initial_height
7.0

hup,fixed_up,(node_up(i),i=fixed_up)
7.0
8
9   18   27   36   45   54   63   72

hdown,fixed_down,(node_down(i),i=fixed_down)
2.0
3
46   55   64

nci
20

```

**Figure 7.12** Mesh and data for first Program 7.3 example

variable converged. The subroutine sets converged to .TRUE. if the solution has converged and to .FALSE. if another iteration is required.

The first example shown in Figure 7.12 is of a vertical-sided dam. The free surface described by nodes 1 through 9 at the top of the mesh is initially assumed to be horizontal. The initial data relates to the number of elements in the  $x$ - (nxe) and  $y$ - (nye) directions,

and this is followed by the tolerance `tol`, set to 0.01 and the iteration ceiling limit, set to 20. The dam is homogeneous in this example, so `np_types` is set to 1. The permeabilities in the  $x$ - and  $y$ -directions are read followed by the  $x$ -coordinates of the bottom `bottom_width` and top nodes `top_width` of the starting mesh. The next line of data reads the initial height of the horizontal free surface `initial_height`, which in this example is set to 7.0. The value of the upstream total head value of `hup=7.0` is then read, followed by the number of nodes to be fixed on the upstream side `fixed_up` and their node numbers `node_up`. Similarly on the downstream side, the total head is set to `hdown=2.0` followed by `fixed_down` and `node_down`. The final data `nci` relates to the number of contour intervals to be plotted.

The output for this example is shown in Figure 7.13. The PostScript output files `*.msh` and `*.con` shown in Figures 7.14 and 7.15, respectively, give the deformed mesh at convergence and a contour map of total head values.

The case of free surface flow through a dam with vertical faces is a classical problem for which the Dupuit formula (see, e.g., Verruijt, 1969) predicts a flow rate given by

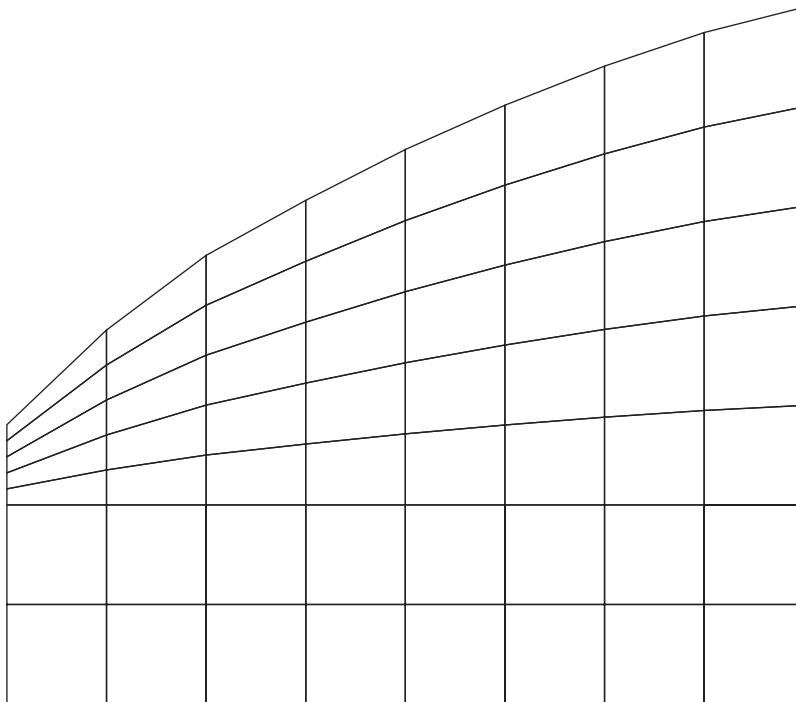
$$Q = \frac{k(H_1^2 - H_2^2)}{2D} \quad (7.3)$$

```
There are      72  equations and the skyline storage is     703
Node  Total Head   Flow rate
  1  0.2759E+01  0.8674E-18
  2  0.3784E+01 -0.2602E-17
  3  0.4498E+01 -0.1301E-17
  4  0.5062E+01 -0.1301E-17
  5  0.5571E+01  0.0000E+00
  6  0.6016E+01  0.1301E-17
  7  0.6409E+01  0.3036E-17
  8  0.6746E+01 -0.2602E-17
  9  0.7000E+01  0.1105E-03
 10  0.2645E+01 -0.3100E-04
.
.
.
 65  0.2840E+01 -0.4337E-18
 66  0.3619E+01 -0.4337E-18
 67  0.4310E+01  0.1409E-17
 68  0.4925E+01 -0.7589E-18
 69  0.5485E+01  0.1518E-17
 70  0.6009E+01 -0.2819E-17
 71  0.6510E+01 -0.3469E-17
 72  0.7000E+01  0.2432E-03

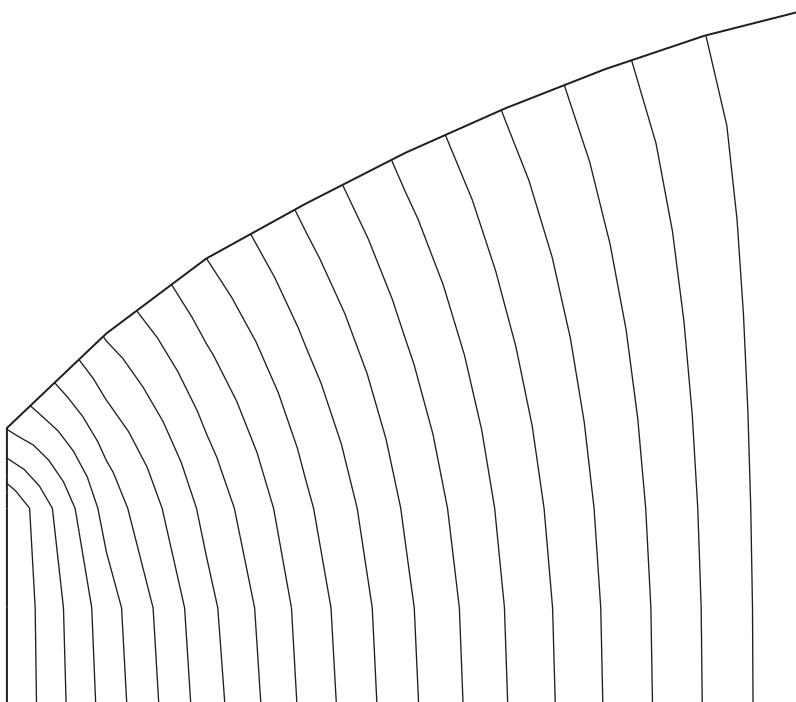
    Inflow        Outflow
-0.2813E-02  0.2813E-02

Converged in 10 iterations
```

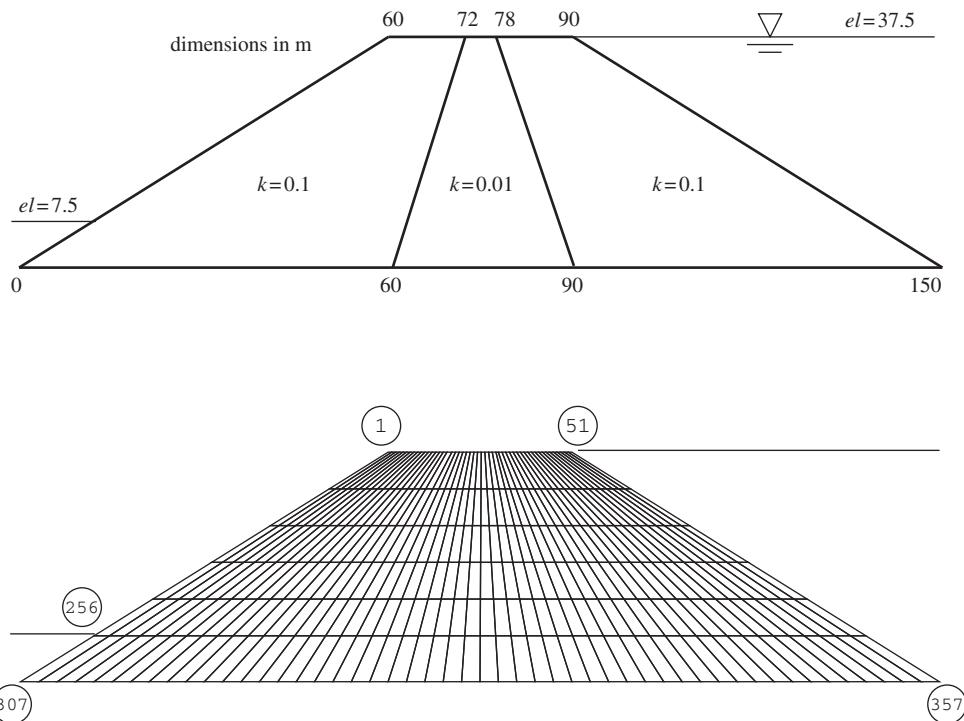
**Figure 7.13** Results from first Program 7.3 example



**Figure 7.14** Computed free surface at convergence in vertical-face dam analysis



**Figure 7.15** Equipotentials at convergence in vertical-face dam analysis



**Figure 7.16** Configuration and mesh for embankment free surface analysis. Second Program 7.3 example

where  $H_1 = 7.0$  m and  $H_2 = 2.0$  m refer to the up- and downstream water elevations,  $k = 0.001$  m/s refers to the permeability (assumed isotropic and homogeneous) and  $D = 8.0$  m refers to the width of the dam. The formula gives a flow rate of  $0.00281 \text{ m}^3/\text{s}/\text{m}$ , which agrees to three significant figures with the computed value.

A second example of an earth dam with sloping sides and a relatively impermeable clay core is presented in Figure 7.16 with data in Figure 7.17. The initial mesh is trapezoidal, and starts with a horizontal free surface set at an elevation of 37.5 m, which is also the height of the starting mesh. The nodes on the upstream face of the dam are also set at a total head of 37.5 m, while the bottom two nodes on the downstream side are fixed at a total head of 7.5 m. The initial mesh is defined by the  $x$ -coordinates of the nodes at the base and the top. There are two property types in this example, so the `etype` data is needed to allocate properties to the elements in the mesh. The middle 10 ‘columns’ of elements represent the clay core.

During the mesh iterations, subroutine `geom_freesurf` ensures that the nodes are constrained to remain on the sloping lines and maintain even spacing in the  $y$ -direction. The output from this example is shown in Figure 7.18, indicating a steady flow of  $0.3335 \text{ m}^3/\text{s}/\text{m}$ . The deformed mesh, which took 11 iterations to converge, is shown in Figure 7.19, which also indicates how the free surface falls rapidly within the clay core.

```

nxe nye tol limit np_types
50      6    0.01  20          2

prop(kx,ky)
0.1    0.1
0.01   0.01

etype
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
.
. etype data for elements 51-250 omitted
.
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

bottom_width, top_width
 0.0     3.0    6.0    9.0   12.0   15.0   18.0
21.0    24.0   27.0   30.0   33.0   36.0   39.0
42.0    45.0   48.0   51.0   54.0   57.0   60.0
63.0    66.0   69.0   72.0   75.0   78.0   81.0
84.0    87.0   90.0   93.0   96.0   99.0   102.0
105.0   108.0  111.0  114.0  117.0  120.0  123.0
126.0   129.0  132.0  135.0  138.0  141.0  144.0  147.0  150.0
60.0    60.6   61.2   61.8   62.4   63.0   63.6
64.2    64.8   65.4   66.0   66.6   67.2   67.8
68.4    69.0   69.6   70.2   70.8   71.4   72.0
72.6    73.2   73.8   74.4   75.0   75.6   76.2
76.8    77.4   78.0   78.6   79.2   79.8   80.4
81.0    81.6   82.2   82.8   83.4   84.0   84.6
85.2    85.8   86.4   87.0   87.6   88.2   88.8   89.4   90.0

initial_height
37.5

hup,(fixed_up,node_up(i),i=fixed_up)
37.5
7
51 102 153 204 255 306 357

hdown,(fixed_down,node_down(i),i=fixed_down)
7.5
2
256 307

nci
20

```

**Figure 7.17** Data for second Program 7.3 example

#### **Program 7.4 General two- (plane) or three-dimensional analysis of steady seepage**

```

PROGRAM p74
!-----
! Program 7.4 General two- (plane) or three-dimensional analysis of steady
! seepage.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,k,loaded_nodes,nci,ndim,nels,neq,nip,nlen, &
nod,nm,np_types; CHARACTER(LEN=15)::argv,element
REAL(iwp)::det,penalty=1.0e20 iwp,zero=0.0 iwp

```

```

!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g_num(:,kdiag(:,node(:),num(:)
REAL(iwp),ALLOCATABLE::coord(:,der(:,deriv(:,disps(:,&
g_coord(:,jac(:,kay(:,kp(:,kv(:,kvh(:,loads(:,&
points(:,prop(:,value(:,weights(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)element,nod,nels,nn,nip,ndim,np_types; neq=nn
ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),etype(nels), &
jac(ndim,ndim),weights(nip),num(nod),g_num(nod,nels),der(ndim,nod), &
deriv(ndim,nod),kp(nod,nod),kay(ndim,ndim),prop(ndim,np_types), &
kdiag(neq),loads(0:neq),disps(0:neq))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)g_coord; READ(10,*)g_num
IF(ndim==2)CALL mesh(g_coord,g_num,argv,nlen,12); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel = 1,nels; num=g_num(:,iel); CALL fkdiag(kdiag,num)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
WRITE(11,'(2(A,I5))')
" There are",neq," equations and the skyline storage is",kdiag(neq)
ALLOCATE(kv(kdiag(neq)),kvh(kdiag(neq))); kv=zero
CALL sample(element,points,weights)
!-----global conductivity matrix assembly-----
elements_2: DO iel=1,nels
  kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
  num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kp=zero
  gauss_pts_1: DO i=1,nip
    CALL shape_der(der,points,i); jac=MATMUL(der,coord)
    det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
    kp=kp+MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
  END DO gauss_pts_1; CALL fspard(kv,kp,num,kdiag)
END DO elements_2; kvh=kv
!-----specify boundary values-----
loads=zero; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
  ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
  READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
  kv(kdiag(node))=kv(kdiag(node))+penalty
  loads(node)=kv(kdiag(node))*value
END IF
!-----equation solution-----
CALL sparin(kv,kdiag); CALL spabac(kv,loads,kdiag)
!-----retrieve nodal net flow rates-----
CALL linmul_sky(kvh,loads,disps,kdiag)
WRITE(11,'(/A)')" Node Total Head Flow rate"
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
disps(0)=zero; WRITE(11,'(/A)')"           Inflow          Outflow"
WRITE(11,'(5X,2E12.4)')
SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero)
IF(ndim==2.AND.nod==4)THEN
  READ(10,*)nci; CALL contour(loads,g_coord,g_num,nci,argv,nlen,13)
END IF
STOP
END PROGRAM p74

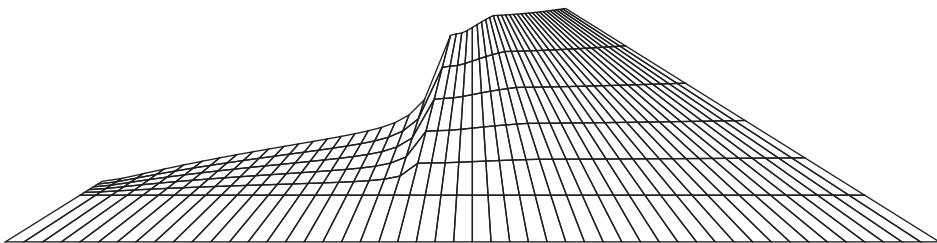
```

There are 357 equations and the skyline storage is 16313

Node	Total Head	Flow rate
1	0.9407E+01	0.8882E-15
2	0.9959E+01	0.0000E+00
3	0.1081E+02	0.0000E+00
4	0.1160E+02	0.8882E-15
5	0.1229E+02	0.4441E-15
6	0.1290E+02	-0.8882E-15
7	0.1345E+02	0.1332E-14
8	0.1396E+02	-0.8882E-15
9	0.1443E+02	0.6661E-15
10	0.1488E+02	-0.2220E-15
.	.	.
350	0.3743E+02	0.1776E-14
351	0.3746E+02	0.3553E-14
352	0.3747E+02	-0.8882E-14
353	0.3749E+02	-0.7105E-14
354	0.3749E+02	-0.1776E-14
355	0.3750E+02	0.3553E-14
356	0.3750E+02	-0.1776E-14
357	0.3750E+02	0.4641E-03
Inflow	Outflow	
-0.3335E+00	0.3335E+00	

Converged in 11 iterations

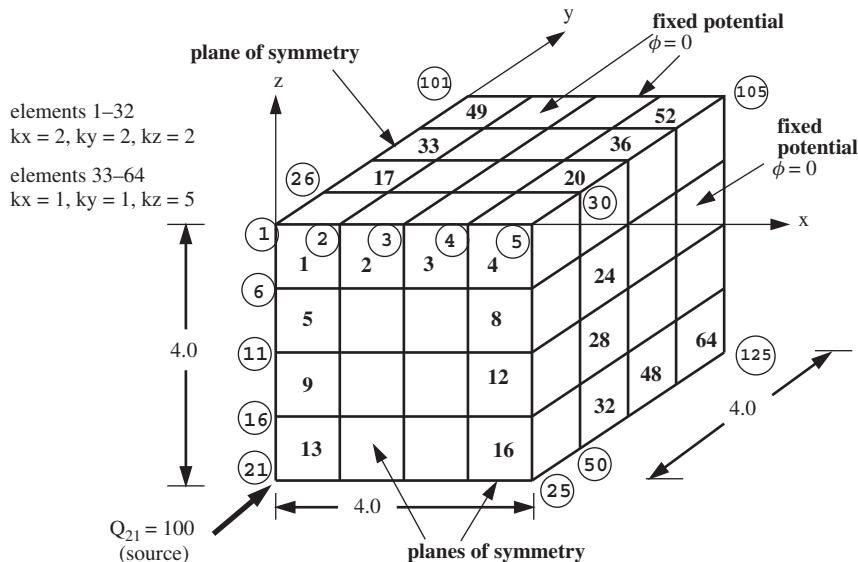
**Figure 7.18** Results from second Program 7.3 example



**Figure 7.19** Computed free surface at convergence in embankment analysis

Program 7.4 can analyse steady confined seepage over any two- or three-dimensional domain with non-homogeneous and anisotropic material properties. The program is very similar to Program 5.4 for general elastic analysis of 2- or 3D, and can use any of the 2- or 3D elements referred to in this book (see Appendix B). The main difference from the programs described previously in this chapter is that this program includes no ‘geometry’ subroutine, so all nodal coordinates `g_coords` and element node numbers `g_num` must be provided as data. In addition, some of the variables that were previously fixed in the declaration statements must now be read as data in order to identify the dimensionality of the problem and the type of element required. There are no variables required by this program that have not already been encountered in earlier programs of this chapter.

A three-dimensional seepage example is shown in Figure 7.20. The model represents one-eighth of a symmetrical cube with a point source of 100 units at its centroid with all outside faces maintained at a total head of zero. Referring to the figure, node numbers



```

element      nod    nels   nn    nip    ndim   np_types
'hexahedron'  8       64     125    8      3          2
prop(kx,ky,kz)
2.0  2.0  2.0      1.0  1.0  5.0

etyp
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

g_coord
0.0  0.0  0.0      1.0  0.0  0.0      2.0  0.0  0.0      3.0  0.0  0.0
.
g_coord data for nodes 5-120 omitted here
.
0.0  4.0  4.0      1.0  4.0  4.0      2.0  4.0  4.0      3.0  4.0  4.0
4.0  4.0  4.0

g_num
6    1    2    7   31   26   27   32      7    2    3    8   32   27   28   33
.
g_num data for elements 3-62 omitted here
.
98   93   94   99  123  118  119  124      99   94   95  100  124  119  120  125

loaded_nodes, (k, loads(k), i=1, loaded_nodes)
1      21   100.0

fixed_freedoms, (node(i), value(i), i=1, fixed_freedoms)
61
1 0.0    2 0.0    3 0.0    4 0.0    5 0.0    10 0.0   15 0.0   20 0.0
.
fixed freedom data omitted here
.
121 0.0   122 0.0   123 0.0   124 0.0   125 0.0

```

Figure 7.20 Mesh and data for Program 7.4 example

are indicated in circles and some of the element numbers have also been included. The example has 125 nodes and 64 elements.

The first line of data identifies the element type `element`, which in this case is a 'hexahedron', the number of nodes on each element `nod`, the number of elements `nels`, the number of nodes in the mesh `nn`, the number of integrating points `nip`, the number of dimensions of the problem `ndim` and the number of property types `np_types`. It may be noted that numerical integration of an 8-node hexahedral element usually requires eight Gauss points (two in each of the three coordinate directions), so `nip` is read as 8.

The problem includes two property types, so the next two lines of data provide the property values for each of the `np_types` groups. A 3D problem (`ndim=3`) such as this requires three permeability terms ( $k_x$ ,  $k_y$  and  $k_z$ ) for each property group. In this example, the first group is applied to elements 1 to 32, which are isotropic with  $k_x = k_y = k_z = 2$ , and the second group to elements 33 to 64, which are anisotropic with  $k_x = k_y = 1$  and  $k_z = 5$ . The `etype` vector then reads the information required to match elements with property groups.

In this chapter we always assume that the principal axes of the permeability tensor coincide with the Cartesian coordinate axes, leading to a diagonal property array `kay`. If this is not the case, the `kay` matrix will be fully populated with off-diagonal terms.

The next data involves the  $x$ ,  $y$ ,  $z$  coordinates of the `nn` nodes in the mesh read into `g_coord`, followed by the node numbers of each of the `nels` elements read into `g_num`. If dealing with a three-dimensional 8-node element, for example, the order in which the node numbers are read must follow the sequence described for that element in Appendix B. Due to the volume of data required in this example, only a few lines of the `g_coord` and `g_num` data are actually shown in Figure 7.20.

There is one source at node 21 equal to 100.0 indicated in the `loaded_nodes` data. All the outside faces of the cube are fixed to zero, which requires 61 `fixed_freedoms` data.

If Program 7.4 is to be applied to a 2D analysis using 4-node quadrilateral elements, a final data input of the number of contour intervals `nci` is also required.

A truncated version of the output from the program is shown in Figure 7.21. The total head is greatest at the central node, and equals about 169. Outflow occurs at all the outside nodes of the mesh where the total head was fixed to zero. For example, the outflow at node number 5 equals -0.3514.

## **Program 7.5 General two- (plane) or three-dimensional analysis of steady seepage. No global conductivity matrix assembly. Diagonally preconditioned conjugate gradient solver**

```
PROGRAM p75
!-----
!  
! Program 7.5 General two- (plane) or three-dimensional analysis of steady  
! seepage. No global conductivity matrix assembly.  
! Diagonally preconditioned conjugate gradient solver.  
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,k,loaded_nodes,nci,ndim, &
nels,neq,nip,nlen,nod,nn,np_types
```

```

REAL(iwp)::alpha,beta,cg_tol,det,one=1.0_iwp,penalty=1.0e20_iwp,up,      &
zero=0.0_iwp
CHARACTER(LEN=15)::argv,element; LOGICAL::cg_converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,),g_num(:,,:),node(:,),num(:)
REAL(iwp),ALLOCATABLE::coord(:, :,),d(:, :),der(:, :,),deriv(:, :,),
diag_precon(:, ),disps(:, ),g_coord(:, :,),jac(:, :,),kay(:, :,),kp(:, :,),
loads(:, ),p(:, ),points(:, :,),prop(:, :,),store(:, ),storkp(:, :, :),u(:, ),
value(:, ),weights(:, ),x(:, ),xnew(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)element,nod,nels,nn,nip,ndim,cg_tol,cg_limit,np_types; neq=nn
WRITE(11,'(A,I5,A)')" There are",neq," equations"
ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),etype(nels),   &
jac(ndim,ndim),weights(nip),num(nod,nel),der(ndim,nod),               &
deriv(ndim,nod),kp(nod,nod),kay(ndim,ndim),prop(ndim,np_types),       &
p(0:neq),loads(0:neq),x(0:neq),xnew(0:neq),u(0:neq),diag_precon(0:neq),&
d(0:neq),disps(0:neq),storkp(nod,nod,nel))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)g_coord
READ(10,*)g_num; IF(ndim==2)CALL mesh(g_coord,g_num,argv,nlen,12)
diag_precon=zero; CALL sample(element,points,weights)
!-----element conductivity integration, storage and preconditioner---
elements_1: DO iel=1,nel
  kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
  num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kp=zero
  gauss_pts_1: DO i=1,nip
    CALL shape_der(der,points,i); jac=MATMUL(der,coord)
    det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
    kp=kp+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
  END DO gauss_pts_1; storkp(:, :,iel)=kp
  DO k=1,nod; diag_precon(num(k))=diag_precon(num(k))+kp(k,k); END DO
END DO elements_1
!-----invert the preconditioner and get starting loads--
loads=zero; READ(10,*)loaded_nodes,(k,loads(k),i=1,loaded_nodes)
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
  ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),                   &
  store(fixed_freedoms))
  READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
  diag_precon(node)=diag_precon(node)+penalty
  loads(node)=diag_precon(node)*value; store=diag_precon(node)
END IF
diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
pcg: DO
  cg_iters=cg_iters+1; u=zero
  elements_2: DO iel=1,nel
    num=g_num(:,iel); kp=storkp(:, :,iel); u(num)=u(num)+MATMUL(kp,p(num))
  END DO elements_2
  IF(fixed_freedoms/=0)u(node)=p(node)*store; up=DOT_PRODUCT(loads,d)
  alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha; loads=loads-u*alpha
  d=diag_precon*loads; beta=DOT_PRODUCT(loads,d)/up; p=d+p*beta
  CALL checon(xnew,x,cg_tol,cg_converged)
  IF(cg_converged.OR.cg_iters==cg_limit)EXIT
END DO pcg
WRITE(11,'(A,I5)')" Number of cg iterations to convergence was",cg_iters

```

```

!-----retrieve nodal net flow rates-----
loads=xnew; disps=zero
elements_3: DO iel=1,nels
    num=g_num(:,iel); kp=storkp(:,:,iel)
    disps(num)=disps(num)+MATMUL(kp,loads(num) )
END DO elements_3; disps(0)=zero
WRITE(11,'(/A)')" Node Total Head Flow rate"
DO k=1,nn; WRITE(11,'(I5,2E12.4)')k,loads(k),disps(k); END DO
WRITE(11,'(/A)')" Inflow Outflow"
WRITE(11,'(5X,2E12.4)')
    SUM(disps,MASK=disps>zero),SUM(disps,MASK=disps<zero)
IF(ndim==2.AND.nod==4)THEN
    READ(10,*)nci; CALL contour(loads,g_coord,g_num,nci,argv,nlen,13)
END IF
STOP
END PROGRAM p75

```

Program 7.5 is the final program in this chapter and is identical to Program 7.4 except for the equation solution strategy, which in this case uses the preconditioned conjugate gradient method with no global matrix assembly. The program is in many ways similar to Program 5.6 in Chapter 5, where the pcg method was first demonstrated.

There are 125 equations and the skyline storage is 3225

Node	Total	Head	Flow rate
1	0.1543E-19	-0.1543E+01	
2	0.2732E-19	-0.2732E+01	
3	0.2028E-19	-0.2028E+01	
4	0.1032E-19	-0.1032E+01	
5	0.3514E-20	-0.3514E+00	
6	0.3319E+01	-0.1332E-14	
7	0.3183E+01	-0.7772E-15	
8	0.2152E+01	0.6661E-15	
9	0.1106E+01	-0.4441E-15	
10	0.1026E-19	-0.1026E+01	
11	0.9727E+01	0.3553E-14	
12	0.6905E+01	0.8882E-15	
13	0.5206E+01	0.4441E-15	
14	0.2141E+01	0.8882E-15	
15	0.2004E-19	-0.2004E+01	
16	0.1121E+02	0.3553E-14	
17	0.2098E+02	0.5995E-14	
18	0.6877E+01	0.4885E-14	
19	0.3154E+01	-0.1998E-14	
20	0.2684E-19	-0.2684E+01	
21	0.1689E+03	0.1000E+03	
.	.	.	
120	0.2361E-20	-0.2361E+00	
121	0.5872E-20	-0.5872E+00	
122	0.8971E-20	-0.8971E+00	
123	0.7082E-20	-0.7082E+00	
124	0.3505E-20	-0.3505E+00	
125	0.1255E-20	-0.1255E+00	
	Inflow	Outflow	
	0.1000E+03	-0.1000E+03	

**Figure 7.21** Results from Program 7.4 example

All of the elements are looped in order to compute their conductivity matrices, which are stored in the array `storkc` for use later in the `pcg` solution algorithm. This loop, called `elements_1`, also builds the preconditioning matrix which is simply the inverse of the diagonal terms in what would have been the assembled global conductivity matrix. The preconditioning matrix (stored as a vector) is called `diag_precon`. The section commented ‘`pcg` equation solution’ carries out the vector operations described in equations (3.22) within the iterative loop labelled `pcg`. The matrix–vector multiply needed in the first of (3.22) is done using (3.23). The node number vector `num` ‘gathers’ the appropriate components of `p`, to be multiplied by the element conductivity matrix `kc` retrieved from `storkc`. Similarly, the vector `num` ‘scatters’ the result of the matrix–vector multiply to appropriate locations in `u`. A tolerance `cg_tol` enables the iterations to be stopped when successive solutions are ‘close enough’, but since `pcg` is a loop which might carry on ‘forever’, an iteration ceiling, `cg_limit`, is specified also. The nodal net flow rates are accumulated from element-by-element matrix–vector products in the loop called `elements_3`.

The problem given in Figure 7.20 is solved once more using Program 7.5 with the data given in Figure 7.22. The only additional data are `cg_tol` and `cg_limit`, the pcg convergence tolerance and iteration ceiling, respectively. The output in Figure 7.23 is identical to that obtained previously using an assembly strategy in Figure 7.21, apart from the additional comment indicating that the pcg algorithm took 18 iterations to converge.

**Figure 7.22** Mesh and data for Program 7.5 example

```

There are 125 equations
Number of cg iterations to convergence was 18

Node Total Head Flow rate
 1 0.0000E+00 -0.1543E+01
 2 0.0000E+00 -0.2732E+01
 3 0.0000E+00 -0.2028E+01
 4 0.0000E+00 -0.1032E+01
 5 0.0000E+00 -0.3514E+00
 6 0.3319E+01 0.1305E-05
 7 0.3183E+01 0.7655E-06
 8 0.2152E+01 -0.1016E-05
 9 0.1106E+01 -0.1038E-05
10 0.0000E+00 -0.1026E+01
11 0.9727E+01 -0.8685E-06
12 0.6905E+01 0.1006E-06
13 0.5206E+01 0.5596E-06
14 0.2141E+01 -0.1743E-05
15 0.0000E+00 -0.2004E+01
16 0.1121E+02 0.2077E-05
17 0.2098E+02 -0.1051E-05
18 0.6877E+01 0.7030E-06
19 0.3154E+01 -0.1291E-05
20 0.0000E+00 -0.2684E+01
21 0.1689E+03 0.1000E+03

.
.

120 0.0000E+00 -0.2361E+00
121 0.0000E+00 -0.5872E+00
122 0.0000E+00 -0.8971E+00
123 0.0000E+00 -0.7082E+00
124 0.0000E+00 -0.3505E+00
125 0.0000E+00 -0.1255E+00

Inflow      Outflow
0.1000E+03 -0.1000E+03

```

**Figure 7.23** Mesh and data for Program 7.5 example

## 7.2 Glossary of variable names

### Scalar integers:

cg_iters	pcg iteration counter
cg_limit	pcg iteration ceiling
fixed_down	number of nodes on downstream side
fixed_freedoms	number of fixed total heads
fixed_seep	number of nodes on seepage surface
fixed_up	number of nodes on upstream side
i_iel	simple counters
iters	counts free surface iterations
iwp	SELECTED_REAL_KIND(15)
k	simple counter
limit	iteration ceiling
loaded_nodes	number of fixed source/sink nodes
nci	number of contour intervals required
ndim	number of dimensions
nels	number of elements
neq	number of degrees of freedom in the mesh
nlen	maximum number of characters in data file basename

---

nod	number of nodes per element
nn	number of nodes
nprops	number of material properties
np_types	number of different property types
nxe, nye	number of columns and rows of elements

**Scalar reals:**

alpha,beta	$\alpha$ and $\beta$ from equations (3.22)
cg_tol	pcg convergence tolerance
det	determinant of the Jacobian matrix
hdown	fixed total head on downstream side
hup	fixed total head on upstream side
d180	set to 180.0
initial_height	initial height of free surface to start process
one	set to 1.0
penalty	set to $1 \times 10^{20}$
tol	convergence tolerance
up	holds dot product $\{\mathbf{R}\}_k^T \{\mathbf{R}\}_k$ from equations (3.22)
zero	set to 0.0

**Scalar characters:**

argv	holds data file basename
dir	direction of element and node numbering
element	element type
type_2d	type of 2D analysis

**Scalar logicals:**

cg_converged	set to .TRUE. if pcg process has converged
converged	set to .TRUE. if mesh has converged

**Dynamic integer arrays:**

etype	element property types
g_num	node numbers for all elements
kdiag	diagonal term locations
node	nodes with fixed total heads
node_down	nodes fixed on downstream side
node_seep	nodes fixed on downstream seepage surface
node_up	nodes fixed on upstream side
num	element node numbers

**Dynamic real arrays:**

angs	angles made by sloping mesh lines to horizontal
bottom_width	x-coordinates of nodes at base of mesh
coord	element nodal coordinates
d	preconditioned rhs vector
der	shape function derivatives with respect to local coordinates
deriv	shape function derivatives with respect to global coordinates
diag_precon	diagonal preconditioner vector

disps	net nodal inflow/outflow
ell	element lengths
fun	shape functions
gc	integrating point coordinates
g_coord	nodal coordinates for all elements
jac	Jacobian matrix
kay	permeability matrix
kc	element conductivity matrix
kv	global conductivity matrix
kvh	copy of kv
loads	global total head vector
oldpot	nodal total head values from previous iteration
p	'descent' vector used in equations (3.22)
points	integrating point local coordinates
prop	element properties
store	stores augmented diagonal terms
storkc	holds element conductivity matrices
surf	holds current total head values of free surface
top_width	$x$ -coordinates of initial nodes at top of mesh
u	vector used in equation (3.22)
value	fixed values of total heads
weights	weighting coefficients
$x$ , $x_{\text{new}}$	'old' and 'new' solution vector
$x_{\text{coords}}$ ,	$x$ - and $y$ -coordinates of mesh layout
$y_{\text{coords}}$	

### 7.3 Exercises

1. Steady seepage is taking place along a 1D pipe containing three porous materials with different permeabilities as indicated in Figure 7.24. The total head difference between the ends of the pipe is 100 units. Use three 1D 'rod' elements to discretise the steady flow problem and hence compute the total head values at the two intermediate locations along the pipe and the steady flow rate through the pipe.

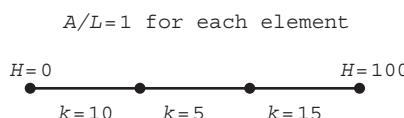


Figure 7.24

Answer: 27.27, 81.82, 272.2

2. The square region in Figure 7.25 has anisotropic conductivity properties and boundary temperatures fixed at the values indicated. Estimate the steady-state temperature at point A.

Answer: 68.3

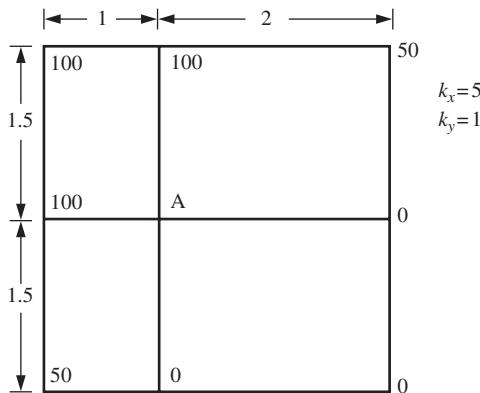


Figure 7.25

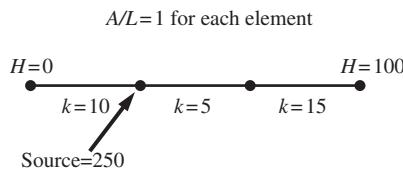


Figure 7.26

3. Steady seepage including a source (positive) is taking place along a 1D pipe containing three porous materials with different permeabilities as indicated in Figure 7.26. Use three 1D ‘rod’ elements to discretise the steady flow problem and hence compute the potential values at the two intermediate locations along the pipe and the net inflow/outflow through the system.

Answer: 45.45, 86.36, 454.5

4. Compute the total head and flow rates at all the nodes in the steady flow problem shown in Figure 7.27.

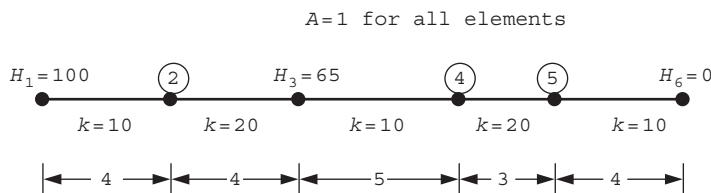


Figure 7.27

Answer: 
$$\begin{cases} H_2 \\ H_4 \\ H_5 \end{cases} = \begin{cases} 76.67 \\ 34.05 \\ 24.76 \end{cases} \quad \begin{cases} Q_1 \\ Q_3 \\ Q_6 \end{cases} = \begin{cases} 58.3 \\ 3.6 \\ -61.9 \end{cases}$$

5. Steady seepage is taking place through the square block of anisotropic porous material with the external source and boundary total head values shown in Figure 7.28. Measurements indicate that  $H_1 = 42.967$  and  $H_2 = -1.535$ . Compute the anisotropic conductivity properties  $k_x$  and  $k_y$ .

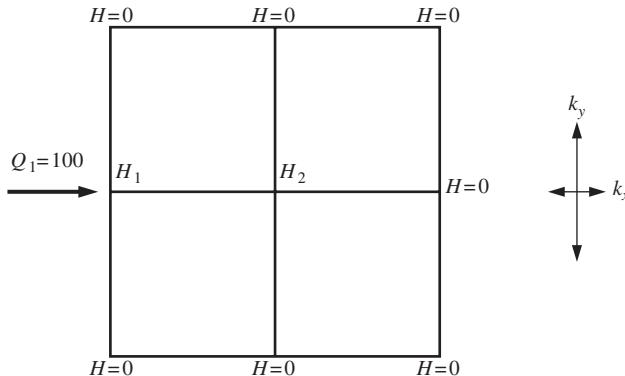


Figure 7.28

Answer:  $k_x = 1$ ,  $k_y = 2.5$

6. 1D steady seepage is taking place down a pipe as shown in Figure 7.29. Compute all the head and net flow rates at the nodes.

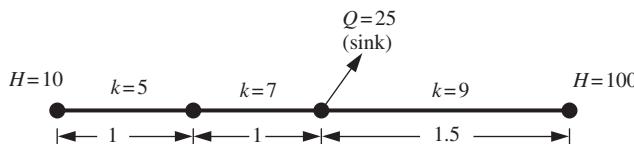


Figure 7.29

Answer:  $H_2 = 43.69$ ,  $H_3 = 67.76$ ,  $Q_1 = -168.45$ ,  $Q_4 = 193.44$

7. Steady heat flow is taking place over the square region shown in Figure 7.30 with the given boundary conditions. All elements have the same conductivity matrix, given by

$$[\mathbf{k}_c] = \begin{bmatrix} 4 & -2.5 & -2 & 0.5 \\ -2.5 & 4 & 0.5 & -2 \\ -2 & 0.5 & 4 & -2.5 \\ 0.5 & -2 & -2.5 & 4 \end{bmatrix}$$

Solve for the central temperature  $T$ .

Answer:  $T = 62.5$

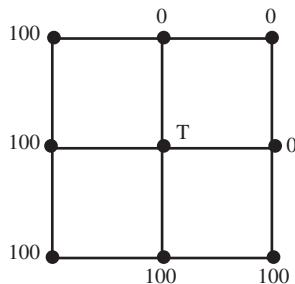


Figure 7.30

8. Use Program 7.2 to estimate the flow rate under the impermeable dam shown in Figure 7.31.

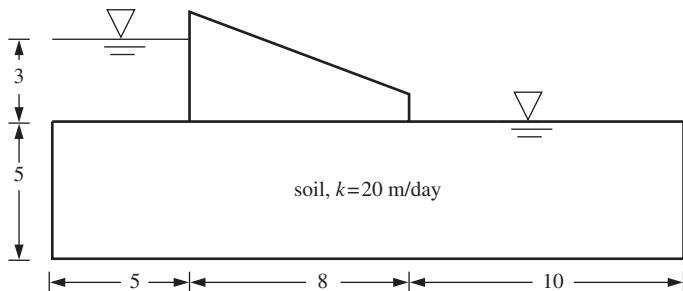


Figure 7.31

Answer:  $Q \approx 25 \text{ m}^3/\text{day}/\text{m}$

9. Use Program 7.4 to estimate the flow rate and exit hydraulic gradient due to seepage beneath the single sheet pile wall shown in Figure 7.32.

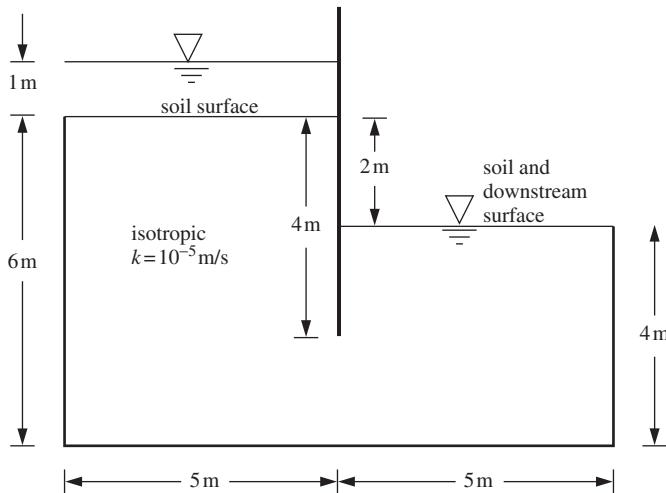


Figure 7.32

Answer: Using 50 square elements of side length 1 unit,  $Q \approx 1.3 \times 10^{-5} \text{ m/s}^3/\text{m}$ ,  $i_e \approx 0.42$

10. Use Program 7.3 to estimate the flow rate due to free surface flow through the symmetric homogeneous embankment shown in Figure 7.33.

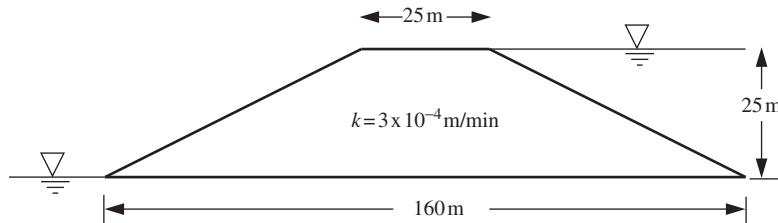


Figure 7.33

Answer:  $Q \approx 12 \times 10^{-4} \text{ m}^3/\text{min/m}$

11. Derive the element conductivity matrix for a square 4-node element suitable for solving Laplace's equation for an isotropic material of permeability  $k$ .

$$\text{Answer: } [\mathbf{k}_e] = \frac{k}{6} \begin{bmatrix} 4 & -1 & -2 & -1 \\ -1 & 4 & -1 & -2 \\ -2 & -1 & 4 & -1 \\ -1 & -2 & -1 & 4 \end{bmatrix}$$

12. Using the matrix from the previous question, assemble the global conductivity matrix for the heat conduction problem shown in Figure 7.34 and hence solve for the steady-state internal temperatures.

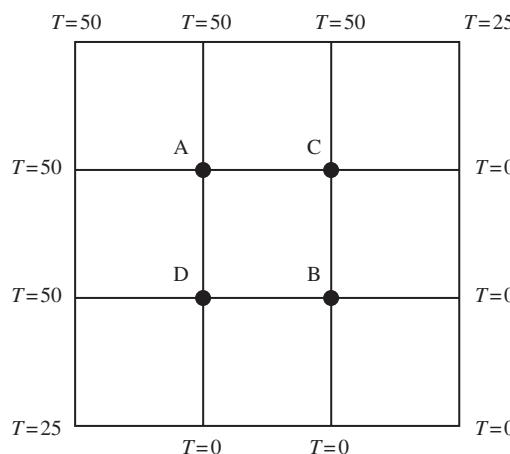


Figure 7.34

Answer:  $T_A = 38.89$ ,  $T_B = 11.11$ ,  $T_C = T_D = 25.00$

13. Derive the conductivity matrix of a 3-node, right-angled isosceles triangular element suitable for discretisation of Laplace's equation. Use your element to estimate the steady-state value of the potential at the central node of the mesh with the boundary conditions given in Figure 7.35.

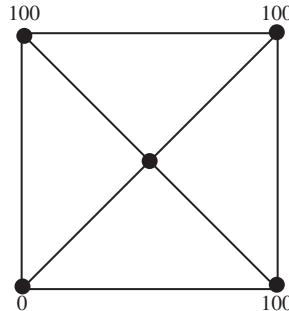


Figure 7.35

Answer: 75.0

14. A square 4-node plane element of unit side length and permeability is to be used in the solution of Laplace's equation over a two-dimensional isotropic medium. If the terms of the element conductivity matrix can be expressed in the form

$$k_{ij} = \int_0^1 \int_0^1 f_{ij}(x, y) dx dy, \quad i, j = 1, 2, 3, 4$$

find the function  $f_{14}$  and evaluate  $k_{14}$  explicitly.

Answer:  $f_{14} = -(1 - y)^2 + x(1 - x)$ ,  $k_{14} = -\frac{1}{6}$

15. Steady seepage is taking place through soil as shown in Figure 7.36 under the given total head boundary conditions. The elements are all square and the hatched sides of the flow path are impermeable. Due to symmetry, a 'tied freedom' numbering scheme has been suggested. Use this numbering system to compute the total head at the remaining nodes and the steady flux through the system. The soil has a constant isotropic permeability as indicated. Check your answer using Program 7.4.

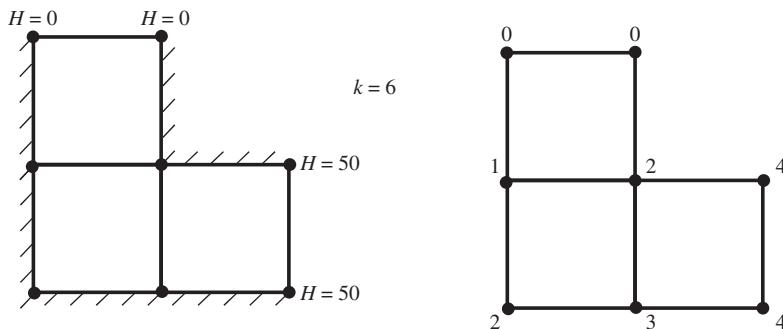


Figure 7.36

Answer:  $H_1 = 17.5$ ,  $H_2 = 25.0$ ,  $H_3 = 32.5$ ,  $Q = 127.5$

## References

- Griffiths DV 1984 Rationalised charts for the method of fragments applied to confined seepage. *Géotechnique* **34**(2), 229–238.
- Griffiths DV 1991 Generalised numerical integration of moments. *Int J Numer Methods Eng* **32**(1), 129–147.
- Verruijt A 1970 *Theory of Groundwater Flow*. Macmillan, London.



# 8

## Transient Problems: First Order (Uncoupled)

### 8.1 Introduction

In the previous chapter, programs for the solution of steady-state potential flow problems were described. Typically, Laplace's equation (2.126) was discretised in space into an equilibrium equation (2.127) involving the solution of a set of simultaneous equations. For well-posed problems there are usually no associated numerical difficulties.

When a flow process is transient, or time-dependent, the simplest extension of equation (2.126), or reduction of the Navier–Stokes equations, is provided by equations like (2.134). There is still a single dependent variable (e.g., potential), and so the analysis is ‘uncoupled’. After discretisation in space, a typical element equation is given by equation (2.138). Problems such as these lead to a set of first-order ordinary differential equations in time, the solution of which is no longer a simple numerical task for large numbers of elements.

Some of the many solution techniques available were described in Chapter 3. These included ‘implicit’ ( $\theta > 0$ ) and ‘explicit’ ( $\theta = 0$ ) methods described by equations (3.100) and (3.104) and by the structure charts of Figures 3.21 and 3.22. Programs 8.1 and 8.2 analyse the 1D consolidation equation using an ‘implicit’ algorithm and Program 8.3 uses an ‘explicit’ algorithm that avoids the need for global matrix assembly. Programs 8.4 and 8.5 extend this approach to 2D. In the case of Program 8.5, an ‘element-by-element’ pre-conditioned conjugate gradient approach is used. Program 8.6 demonstrates an ‘explicit’ solution strategy to a 2D transient problem, and Program 8.7 uses an ‘element-by-element’ operator splitting method. Program 8.8 returns to the ‘implicit’  $\theta$  methods allowing analysis of 2- or 3D transient problems over a general finite element mesh. Programs 8.9 and 8.10 solve the diffusion–convection equation (2.139) in 2D using, respectively, ‘transformed’ and ‘untransformed’ analyses. The final Program 8.11 demonstrates finite element implementation of the convection boundary conditions as discussed in Section 3.6.1.

## Program 8.1 One-dimensional transient (consolidation) analysis using 2-node ‘line’ elements. Implicit time integration using the ‘theta’ method

```

PROGRAM p81
!-----
! Program 8.1 One dimensional consolidation analysis using 2-node "rod"
!           elements. Implicit time integration using the "theta" method.
!-----
USE main; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,j,nels,neq,nlen,nod=2,npri,nprops=1,
     np_types,nres,nstep,ntime
REAL(iwp)::at,a0,dtim,penalty=1.0e20_iwp,pt5=0.5_iwp,theta,time,
     zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),node(:),num(:),kdiag(:)
REAL(iwp),ALLOCATABLE::bp(:),ell(:,!),kc(:,!),kv(:,!),loads(:,!),newlo(:,!),&
     mm(:,!),press(:,!),prop(:,!),storbp(:,!),value(:,!)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,np_types; neq=nels+1
ALLOCATE(num(nod),etype(nels),kc(nod,nod),mm(nod,nod),press(0:neq),&
     prop(nprops,np_types),ell(nels),kdiag(neq),loads(0:neq),newlo(0:neq))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
READ(10,*)dtim,nstep,theta,npri,nres,ntime; kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    num=(/iel,iel+1/); CALL fkdiag(kdiag,num)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq))); bp=zero; kv=zero
WRITE(11,'(2(a,i5))') &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global conductivity and "mass" matrix assembly-----
elements_2: DO iel=1,nels; num=(/iel,iel+1/)
    CALL rod_km(kc,prop(1,etype(iel)),ell(iel)); CALL rod_mm(mm,ell(iel))
    CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
END DO elements_2; kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta
!-----specify initial and boundary values-----
loads(0)=zero; READ(10,*)loads(1:); a0=zero
DO iel=1,nels; a0=a0+pt5*ell(iel)*(loads(iel)+loads(iel+1)); END DO
READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)then
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),&
             storbp(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
END IF
!-----factorise equations-----
CALL sparin(bp,kdiag)
!-----time stepping loop-----
WRITE(11,'(/a,i3,a)')" Time      Uav      Pressure (node",nres,")"
WRITE(11,'(3e12.4)')0.0,0.0,loads(nres)
timesteps: DO j=1,nstep

```

```

time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag)
IF(fixed_freedoms/=0) newlo(node)=storbp*value
CALL spabac(bp,newlo,kdiag); loads=newlo; at=zero
DO iel=1,nels; at=at+pt5*ell(iel)*(loads(iel)+loads(iel+1)); END DO
IF(j==ntime)press(1:)=loads(1:)
IF(j/npri*npri==j)WRITE(11,'(3e12.4)')time,(a0-at)/a0,loads(nres)
END DO timesteps
WRITE(11,'(/a,e10.4,a)')" Depth Pressure (time=",ntime*dtim,")"
WRITE(11,'(3e12.4)')0.0,press(1)
WRITE(11,'(2e12.4)')(SUM(ell(1:i)),press(i+1),i=1,nels)
STOP
END PROGRAM p81

```

In the absence of sources or sinks, equation (3.100) reduces to

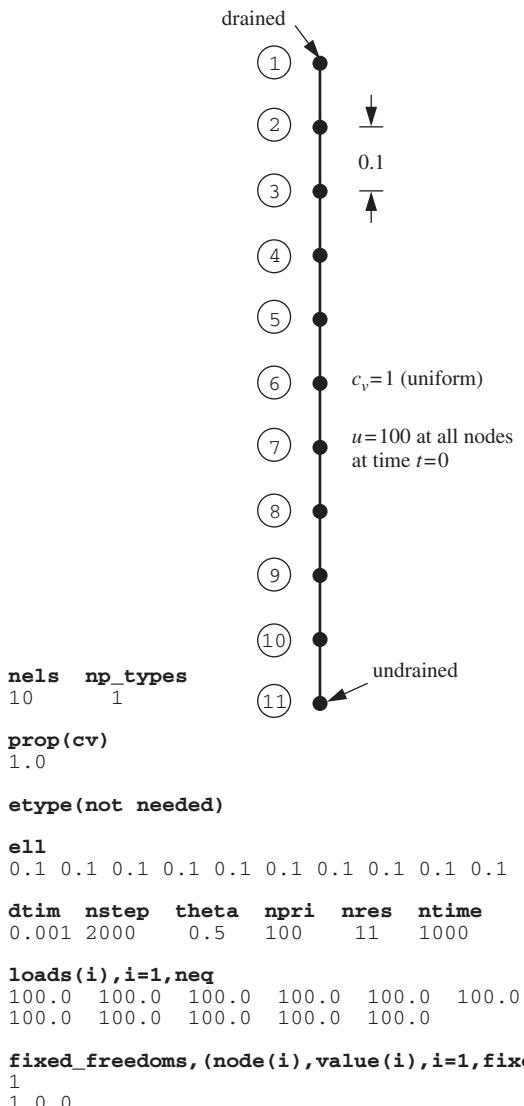
$$([\mathbf{M}_m] + \theta \Delta t [\mathbf{K}_c])\{\Phi\}_1 = ([\mathbf{M}_m] - (1 - \theta) \Delta t [\mathbf{K}_c])\{\Phi\}_0 \quad (8.1)$$

where the element conductivity  $[\mathbf{k}_c]$  and ‘mass’  $[\mathbf{m}_m]$  matrices ( $\mathbf{k}_c$  and  $\mathbf{m}_m$ , respectively in programming terminology) have been assembled into their global counterparts  $[\mathbf{K}_c]$  and  $[\mathbf{M}_m]$  ( $\mathbf{kv}$  and  $\mathbf{bp}$ ). After some manipulations, the left-hand-side matrix  $([\mathbf{M}_m] + \theta \Delta t [\mathbf{K}_c])$  is formed (called  $\mathbf{bp}$ ) and the fixed boundary conditions are implemented using the ‘stiff spring’ or ‘penalty’ strategy. The matrix is then factorised by subroutine `sparin`, the initial conditions are read into  $\{\Phi\}_0$  (`loads`) and the right-hand-side matrix-by-vector product  $([\mathbf{M}_m] - (1 - \theta) \Delta t [\mathbf{K}_c])\{\Phi\}_0$  (`newlo`) computed. Within each time step, all that is then required to advance the solution is a matrix-by-vector multiplication on the right-hand side of equation (8.1), followed by a forward and backward substitution. The final section of the program consists of the time-stepping loop completed `nstep` times. The matrix-by-vector multiplication is carried out by `linmul_sky`, and forward and backward substitution by `spabac`. The process is described by the structure chart in Figure 3.21. Note, however, that the matrix-by-vector multiplication on the right-hand side could be done using element-by-element summation, avoiding storage of one large matrix.

Figure 8.1 shows a string of 10 elements attached end to end, representing a 1D layer of saturated soil with a total depth of 1.0. The layer is subjected to a uniform initial excess pore pressure distribution of 100.0 and is drained at the top only, so the maximum drainage path  $D = 1$ . The material property required in this analysis is the coefficient of consolidation  $c_v$  (analogous to  $EA$  in Program 4.1 and  $kA$  in Program 7.1). The objective of the analysis is to compute the excess pore pressure distribution and the average degree of consolidation  $U_{av}$  as a function of time, where  $U_{av}$  is the defined as the proportion of the initial excess pore pressure distribution that has dissipated after time  $t$ .

The data involve reading the number of elements `nels=10` and the number of property types `np_types=1`. In this case the layer is uniform, so with `np_types=1` the `etype` data is not needed. The 10 element lengths `ell` are read, and in this example are all the same length and equal to 0.1.

The next line of data refers to time stepping and the output parameters. The three time-stepping parameters are the calculation time step `dtime`, read as 0.001, the number of calculation time steps required `nstep`, read as 2000 and the time-integration parameters `theta`, read next as 0.5. The choice of  $\theta = 0.5$  is often referred to as the ‘Crank–Nicolson’ method of time integration. The three output control parameters are the output frequency parameter `npri`, read as 100, the node at which a time history is



**Figure 8.1** Mesh and data for Program 8.1 example

required `nres`, read as 11 (corresponding to the bottom node at the impermeable boundary) and the time step at which a spatial distribution of excess pore pressure is required `ntime`, read as 1000 and corresponding to  $t = 1.0$ .

The next data read into `loads` gives the initial excess pore pressure at the 11 nodes at  $t = 0.0$ , which in this example is uniform and equal to 100.0. The final data gives the drainage boundary conditions. In this example, one node (node 1) is a drainage boundary, so its excess pore pressure is fixed equal to zero. Users are invited to try other initial conditions and boundary conditions (e.g., linear variation of excess pore pressure, double drainage, etc.).

There are 11 equations and the skyline storage is 21

Time	$U_{av}$	Pressure (node 11)
0.0000E+00	0.0000E+00	0.1000E+03
0.1000E+00	0.3567E+00	0.9525E+02
0.2000E+00	0.5041E+00	0.7743E+02
0.3000E+00	0.6134E+00	0.6079E+02
0.4000E+00	0.6981E+00	0.4751E+02
0.5000E+00	0.7643E+00	0.3711E+02
0.6000E+00	0.8159E+00	0.2898E+02
0.7000E+00	0.8562E+00	0.2263E+02
0.8000E+00	0.8877E+00	0.1767E+02
0.9000E+00	0.9123E+00	0.1380E+02
0.1000E+01	0.9315E+00	0.1078E+02
0.1100E+01	0.9465E+00	0.8417E+01
0.1200E+01	0.9582E+00	0.6574E+01
0.1300E+01	0.9674E+00	0.5134E+01
0.1400E+01	0.9745E+00	0.4009E+01
0.1500E+01	0.9801E+00	0.3131E+01
0.1600E+01	0.9845E+00	0.2445E+01
0.1700E+01	0.9879E+00	0.1909E+01
0.1800E+01	0.9905E+00	0.1491E+01
0.1900E+01	0.9926E+00	0.1165E+01
0.2000E+01	0.9942E+00	0.9094E+00

Depth	Pressure (time=0.1000E+01)
0.0000E+00	-0.1967E-21
0.1000E+00	0.1686E+01
0.2000E+00	0.3331E+01
0.3000E+00	0.4893E+01
0.4000E+00	0.6335E+01
0.5000E+00	0.7621E+01
0.6000E+00	0.8720E+01
0.7000E+00	0.9604E+01
0.8000E+00	0.1025E+02
0.9000E+00	0.1065E+02
0.1000E+01	0.1078E+02

**Figure 8.2** Results from Program 8.1 example

The results given in Figure 8.2 show the time history of the excess pore pressure at node 11 and the average degree of consolidation ( $U_{av}$ ), every 100 time steps up to  $t = 2.0$ . The lower part of the output file gives the excess pore pressure with depth corresponding to  $t = 1.0$ .

With the maximum drainage path ( $D$ ) and the coefficient of consolidation ( $c_v$ ) both set equal to unity in this example, the dimensionless time factor  $T$  equals real time, thus  $T = c_v t / D^2 = t$ .

Figure 8.3 gives a plot of the computed  $U_{av}$  vs.  $T$ , showing excellent agreement with the series solution (e.g., Taylor, 1948).

### 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

PROGRAM p82

```
!-----
! Program 8.2 One dimensional consolidation analysis
! (settlement and excess pore pressure) using 2-node "line" elements.
! Implicit time integration using the "theta" method.
!-----
```

```

USE main; IMPLICIT NONE ; INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,j,nels,neq,nlen,nod=2,npri,nprops=2,          &
np_types,nres,nstep,ntime
REAL(iwp)::at,a0,dtim,gamw,penalty=1.0e20_iwp,pt5=0.5_iwp,sc,sl,theta,      &
time,uav,uavs,zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),kdiag(:,node(:),num(:)
REAL(iwp),ALLOCATABLE::bp(:,ell(:,kc(:, :,kv(:, :,loads(:,mm(:, :,newlo(:,press(:,prop(:, :,storbp(:,value(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,np_types; neq=nels+1
ALLOCATE(num(nod),etype(nels),kc(nod,nod),mm(nod,nod),press(0:neq),        &
prop(nprops,np_types),ell(nels),kdiag(neq),loads(0:neq),newlo(0:neq))
READ(10,*)prop,gamw ! prop(1,:)=k, prop(2,:)=mv
etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)ell,dtim,nstep,theta,npri,nres,ntime; kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    num=(/iel,iel+1/); CALL fkdiag(kdiag,num)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq))); bp=zero; kv=zero
WRITE(11,'(2(A,I5))')                                     &
" There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global conductivity and "mass" matrix assembly-----
elements_2: DO iel=1,nels; num=(/iel,iel+1/)
    CALL rod_km(kc,prop(1,etype(iel))/gamw,ell(iel))
    CALL rod_mm(mm,ell(iel)*prop(2,etype(iel)))
    CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
END DO elements_2; kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta
!-----specify initial and boundary values-----
READ(10,*)loads(1:); loads(0)=zero; a0=zero; sc=zero
DO iel=1,nels
    a0=a0+pt5*ell(iel)*(loads(iel)+loads(iel+1))
    sc=sc+pt5*ell(iel)*prop(2,etype(iel))*(loads(iel)+loads(iel+1))
END DO; READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)then
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),           &
    storbp(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
END IF
!-----factorise equations-----
CALL sparin(bp,kdiag)
!-----time stepping loop-----
WRITE(11,'(/2A,I4,A)')"      Time      U_av      U_av_s",           &
"      Settlement Pressure (node",nres,")"
WRITE(11,'(6E12.4)')0.0,0.0,0.0,0.0,loads(nres)
timesteps: DO j=1,nstep
    time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag)
    IF(fixed_freedoms/=0)newlo(node)=storbp*value
    CALL spabac(bp,newlo,kdiag); loads=newlo; at=zero; sl=zero
    DO iel=1,nels
        at=at+pt5*ell(iel)*(loads(iel)+loads(iel+1))
        sl=sl+pt5*ell(iel)*prop(2,etype(iel))*(loads(iel)+loads(iel+1))
    END DO

```

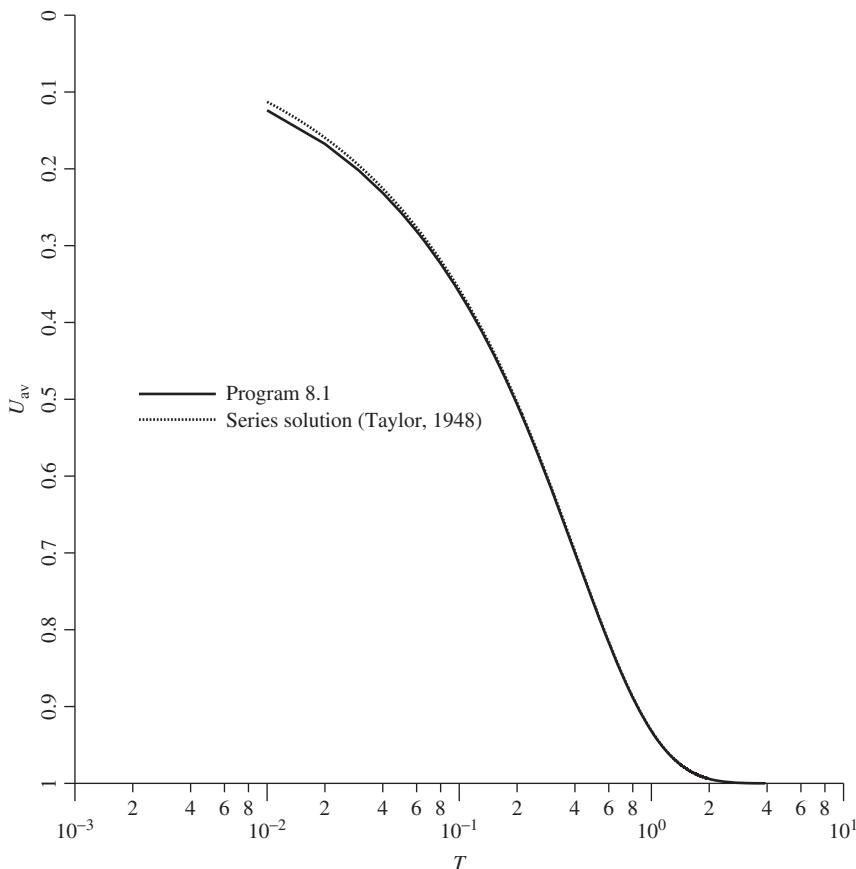
```

uav=(a0-at)/a0; uavs=(sc-sl)/sc; IF(j==ntime)press(1:)=loads(1:)
IF(j/npri*npri==j)WRITE(11,'(6E12.4)')time,uav,uavs,sc-sl,loads(nres)
END DO timesteps
WRITE(11,'(/A,E10.4,A)')" Depth Pressure (time=",ntime*dtim,")"
WRITE(11,'(3E12.4)')0.0,press(1)
WRITE(11,'(2E12.4)')(SUM(ell(1:i)),press(i+1),i=1,nels)
STOP
END PROGRAM p82

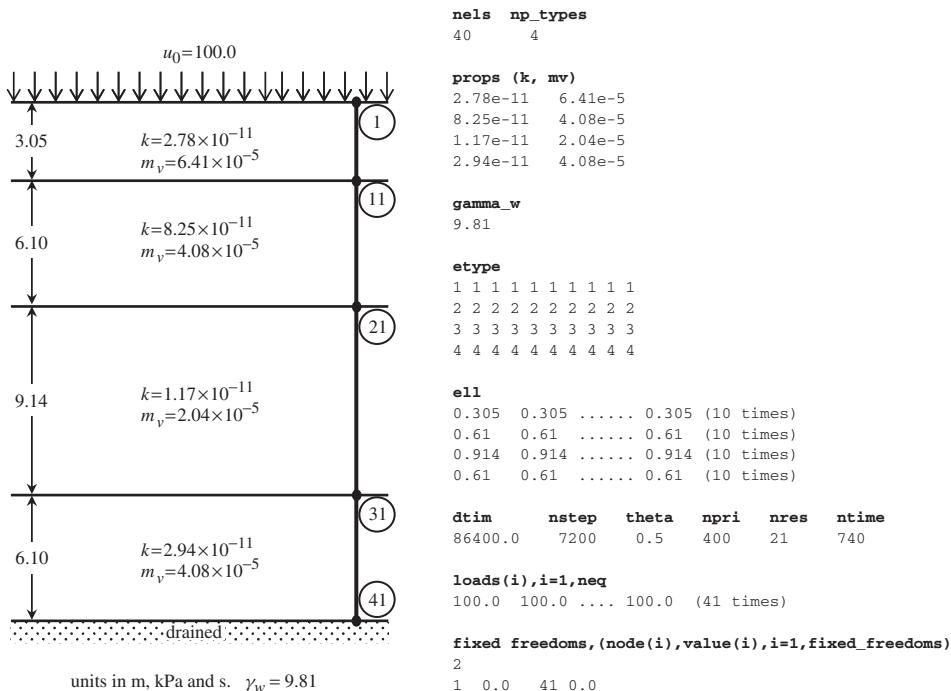
```

The previous program demonstrated a general 1D first-order transient analysis where the material was described by a single diffusion property, namely the coefficient of consolidation in a soil mechanics problem, or perhaps a thermal diffusivity in heat flow analysis. As discussed in Section 2.17.2, however, for predictions of time-dependent settlement of layered systems involving variable compressibility, the coefficients of consolidation should be split into their more fundamental constituents, namely permeability  $k$  and compressibility  $m_v$ . In this program, therefore,  $k$  and  $m_v$  are read in for each layer.

The example problem shown in Figure 8.4 is a double-drained system of total depth  $H = 24.4$  m ( $D = 12.2$  m), with four soil layers as considered by Schiffman and



**Figure 8.3** Comparison of Program 8.1 results with series solution



**Figure 8.4** Mesh and data for Program 8.2 example

Stein (1970). Each layer is discretised into 10 ‘line’ elements, so there is a total of `nels=40` elements. Each layer has different permeability and compressibility properties read into `prop` and the unit weight of water is read as `gamw=9.81`. Since there are `np_types=4` property groups, the `etype` data indicates which property group belongs to which element going from top to bottom. The 40 element lengths are read into `e11`.

As in Figure 8.1, the next line of data provides time-stepping information which calls for 7200 time steps of 86,400 s (equal to 1 day) using the Crank–Nicolson method. Excess pore pressures will be printed every 400 time steps at node 21, which is at the interface between the second and third layers. A spatial distribution of excess pore pressure is called for at the 740th time step.

The next line of data gives the initial excess pore pressure distribution at the 41 nodes, which in this example is again uniform and equal to 100.0. The final data gives the two drainage boundary conditions, which in this case (due to double drainage) requires that the excess pore pressure at the top (node 1) and bottom (node 41) is set to zero.

The truncated results shown in Figure 8.5 give the time history at the desired time output values. The column marked `Uav` is the average degree of consolidation based on excess pore pressure dissipation as defined for Program 8.1, and the column marked `Uavs` is the average degree of consolidation based on settlement; namely, the settlement at time  $t$  divided by the ultimate settlement value after a ‘long’ time. As pointed out by Huang and Griffiths (2010), the values in these two columns are typically different unless  $m_v$  is the same in all layers. The remaining columns marked `Settlement` and `Pressure`

There are 41 equations and the skyline storage is 81

Time	Uav	Uavs	Settlement	Pressure (node 21)
0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.1000E+03
0.3456E+08	0.1327E+00	0.1853E+00	0.1630E-01	0.9991E+02
0.6912E+08	0.1945E+00	0.2621E+00	0.2306E-01	0.9773E+02
0.1037E+09	0.2478E+00	0.3217E+00	0.2830E-01	0.9309E+02
.	.	.	.	.
0.5184E+09	0.6590E+00	0.7043E+00	0.6195E-01	0.4312E+02
0.5530E+09	0.6806E+00	0.7231E+00	0.6361E-01	0.4044E+02
0.5875E+09	0.7009E+00	0.7406E+00	0.6516E-01	0.3792E+02
0.6221E+09	0.7198E+00	0.7571E+00	0.6660E-01	0.3555E+02
.	.	.	.	.
Depth	Pressure (time=0.6394E+08)			
0.0000E+00	-0.2950E-24			
0.3050E+00	0.1032E+02			
0.6100E+00	0.2049E+02			
0.9150E+00	0.3036E+02			
.	.	.	.	.
0.2255E+02	0.4491E+02			
0.2317E+02	0.3090E+02			
0.2378E+02	0.1575E+02			
0.2439E+02	-0.6200E-24			

**Figure 8.5** Results from Program 8.2 example

give, respectively, the surface settlement and the excess pore pressure at the requested node nres=21.

The plot shown in Figure 8.6 gives the dimensionless spatial distribution of excess pore pressure with depth after 740, 2930 and 7195 days, where  $u_o = 100 \text{ kPa}$ . The results are in close agreement with analytical solutions of Schiffman and Stein (1970). The user is invited to run the program with more time steps to show that the top settlement eventually converges on the ultimate value of about 88 mm.

### Program 8.3 One-dimensional consolidation analysis using 2-node 'line' elements. Explicit time integration. Element by element. Lumped mass

```

PROGRAM p83
!-----
! Program 8.3 One dimensional consolidation analysis using 2-node "line"
!           elements. Explicit time integration. Element-by-element.
!           Lumped mass.
!-----
USE main; IMPLICIT NONE; INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,j,nels,neq,nlen,nod=2,npri,nprops=1,      &
          np_types,nres,nstep,ntime
REAL(iwp)::at,a0,dtim,one=1.0_iwp,pt5=0.5_iwp,time,two=2.0_iwp,      &

```

```

zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),node(:,num(:)
REAL(iwp),ALLOCATABLE::ell(:,globma(:,kc(:, :, loads(:, mass(:, mm(:, :, &
newlo(:, press(:, prop(:, :, store_mm(:, :, value(:)

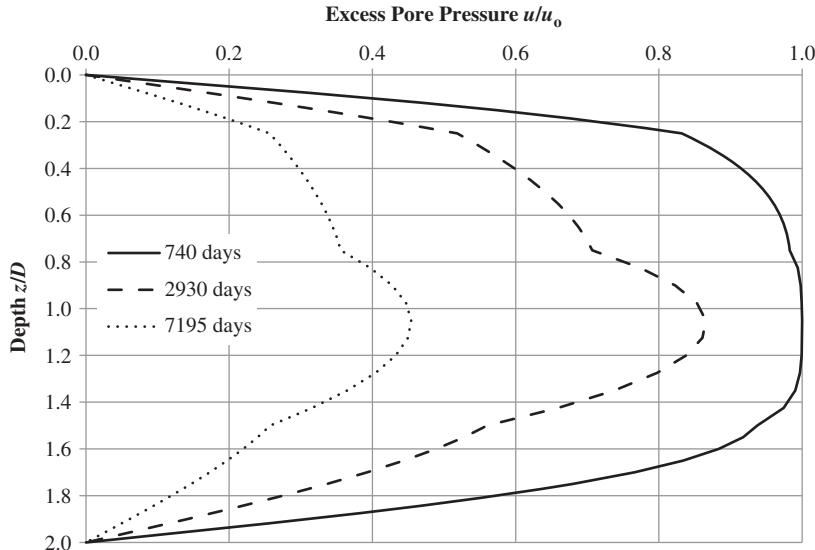
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,np_types; neq=nels+1
ALLOCATE(num(nod),etype(nels),kc(nod,nod),mm(nod,nod),press(0:neq),
prop(nprops,np_types),ell(nels),loads(0:neq),newlo(0:neq),mass(nod),
globma(0:neq),store_mm(nod,nod,nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)ell,dtim,nstep,npri,nres,ntime; globma=zero
WRITE(11,'(2(A,I5))'" There are",neq," equations"
!-----global conductivity and "mass" matrix assembly---
elements_1: DO iel=1,nels
    num=(/iel,iel+1/); CALL rod_km(kc,prop(1,etype(iel)),ell(iel)); mm=zero
    DO i=1,nod; mm(i,i)=ell(iel)/two; mass(i)=ell(iel)/two; END DO
    store_mm(:, :, iel)=mm-kc*dtim; globma(num)=globma(num)+mass
END DO elements_1

!-----specify initial and boundary values-----
READ(10,*)loads(1:); loads(0)=zero
READ(10,*)fixed_freedoms; globma(1:)=one/globma(1:)
IF(fixed_freedoms/=0)then
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
END IF

!-----time stepping loop-----
WRITE(11,'(/A,I3,A)'" Time Uav Pressure (node",nres,")'
WRITE(11,'(3E12.4)')0.0,0.0,loads(nres); a0=zero
DO iel=1,nels; a0=a0+pt5*ell(iel)*(loads(iel)+loads(iel+1)); END DO
timesteps: DO j=1,nstep
    time=j*dtim; newlo=zero
    elements_2: DO iel=1,nels
        num=(/iel,iel+1/); mm=store_mm(:, :, iel)
        newlo(num)=newlo(num)+MATMUL(mm,loads(num))
    END DO elements_2
    newlo(0)=zero; loads=newlo*globma
    IF(fixed_freedoms/=0)loads(node)=value; at=zero
    DO iel=1,nels; at=at+pt5*ell(iel)*(loads(iel)+loads(iel+1)); END DO
    IF(j==ntime)press(1:)=loads(1:)
    IF(j/npri*npri==j)WRITE(11,'(3E12.4)')time,(a0-at)/a0,loads(nres)
END DO timesteps
WRITE(11,'(/A,E10.4,A)'" Depth Pressure (time=",ntime*dtim,")"
WRITE(11,'(3E12.4)')0.0,press(1)
WRITE(11,'(2E12.4)')(SUM(ell(1:i)),press(i+1),i=1,nels)
STOP
END PROGRAM p83

```

The oldest and simplest algorithm for advancing the solution in time uses ‘lumped mass’ and is called an ‘explicit’ method based on (3.104). The method is equivalent to setting the time-stepping parameter  $\theta = 0$  and can be constructed to avoid global matrix assembly.



**Figure 8.6** Excess pore pressure distributions from Program 8.2 example

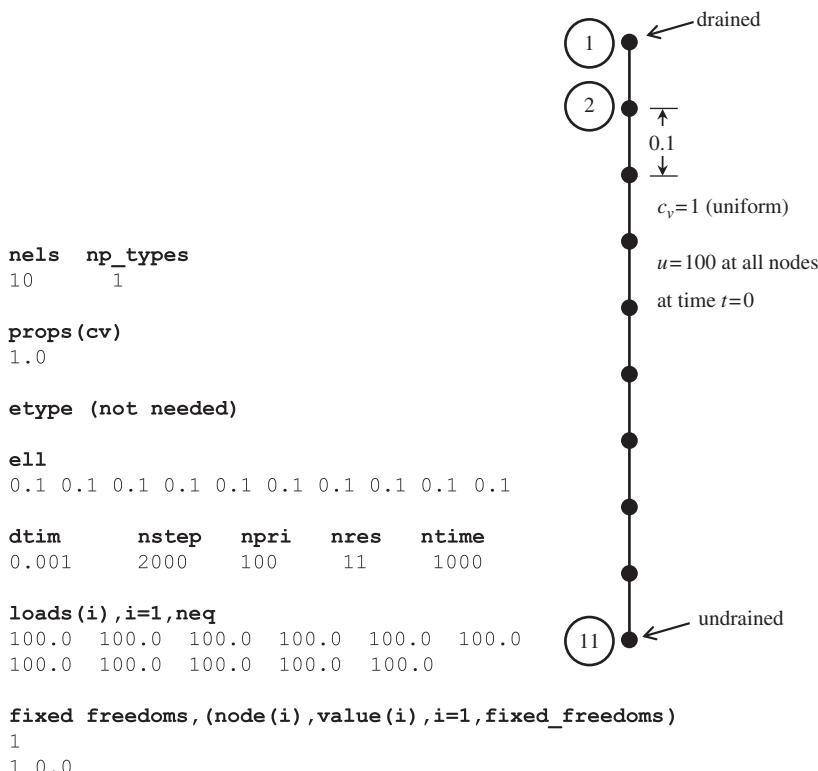
The element integration loop follows the now standard course, but the element matrix ( $[\mathbf{m}_m] - \Delta t [\mathbf{k}_c]$ ) is stored in `store_mm` for each element rather than assembled as would be the case for a traditional implicit technique. Then, in the time-stepping loop, element matrices are recovered from `store_mm` and the product  $([\mathbf{M}_m] - \Delta t [\mathbf{K}_c])\{\Phi\}_0$  formed ‘element-by-element’ using the summation

$$\sum_{i=1}^{nels} ([\mathbf{m}_m] - \Delta t [\mathbf{k}_c])_i \{\phi\}_{0i} \quad (8.2)$$

where  $\{\phi\}_{0i}$  is the appropriate part of  $\{\Phi\}_0$  for element  $i$ . The result of this ‘element-by-element’ product is called `newlo` in programming terminology.

This having been done, the global  $\{\Phi\}_1$  called `loads` is computed by multiplying `newlo` by the inverse of the global mass matrix `globma`. The process is illustrated by the structure chart in Figure 3.22.

The data and example shown in Figure 8.7 are exactly as in Figure 8.1, with the exception that the time-stepping parameter `theta` is absent since it must equal zero. The results shown in Figure 8.8 agree closely with those from the implicit algorithm in Figure 8.2. The disadvantage of explicit algorithms, however, is that they are conditionally stable and rely on the time step being sufficiently small. Figure 8.9 shows the computed excess pore pressure vs. time at node 11 for the example problem using time steps of  $\Delta t = 0.005$  and  $0.0051$ . While the results with  $\Delta t = 0.005$  appear stable, and are essentially the same as those shown in Figure 8.8 with  $\Delta t = 0.001$ , the slightly higher time step leads to an entirely unstable result where the oscillations are growing unboundedly. The critical time step for this analysis is  $0.5 \times (\text{element length})^2$ , or  $0.005$ .



**Figure 8.7** Mesh and data for Program 8.3 example

#### 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

```

PROGRAM p84
!-----
! Program 8.4 Plane or axisymmetric transient analysis using 4-node
!           rectangular quadrilaterals. Mesh numbered in x(r)- or y(z)-
!           direction. Implicit time integration using the "theta"
!           method.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freeoms,i,iel,j,nci,ndim=2,nel,neq,nip=4,nlen,nn,nod=4, &
          npri,np_types,nres,nstep,ntime,nxe,nye
REAL(iwp)::det,dtim,one=1.0_iwp,penalty=1.0e20_iwp,theta,time, &
            zero=0.0_iwp; CHARACTER(len=15)::argv,dir,element='quadrilateral',type_2d
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g_num(:,:,),node(:,),num(:,),kdiag(:)
REAL(iwp),ALLOCATABLE::bp(:, ),coord(:, :, ),der(:, :, ),deriv(:, :, ),fun(:, ), &
  gc(:, ),g_coord(:, :, ),jac(:, :, ),kay(:, :, ),kc(:, :, ),kv(:, ),loads(:, ),newlo(:, ), &

```

```

ntn(:,:,mm(:,:,points(:,:,prop(:,:,storbp(:),value(:),weights(:), &
x_coords(:,y_coords(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,dir,nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim), &
fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod), &
mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod), &
etype(nels),kdiag(neq),loads(0:neq),newlo(0:neq),x_coords(nxe+1), &
y_coords(nye+1),prop(ndim,np_types),gc(ndim))
READ(10,*)prop; etype=1; if(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords
READ(10,*)dtim,nstep,theta,npri,nres,ntime; kdiag=0
! -----loop the elements to set up global geometry and kdiag -----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
    CALL fkdiag(kdiag,num)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq)))
WRITE(11,'(2(a,i5))')                                     &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); bp=zero; kv=zero; gc=one
!-----global conductivity and "mass" matrix assembly-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); if(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
        CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
    END DO gauss_pts
    CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
END DO elements_2; kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta
!-----specify initial and boundary values-----
READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)then
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms), &
              storbp(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
END IF
!-----factorise equations-----
CALL sparin(bp,kdiag)
!-----time stepping loop-----
WRITE(11,'(/a,i3,a)')" Time      Pressure (node",nres,")"
WRITE(11,'(2e12.4)')0.0,loads(nres)
timesteps: DO j=1,nstep
    time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag)
    IF(fixed_freedoms/=0)newlo(node)=storbp*value
    CALL spabac(bp,newlo,kdiag); loads=newlo

```

```

IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
  CALL contour.loads,g_coord,g_num,nci,argv,nlen,13); END IF
  IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
END DO timesteps
STOP
END PROGRAM p84

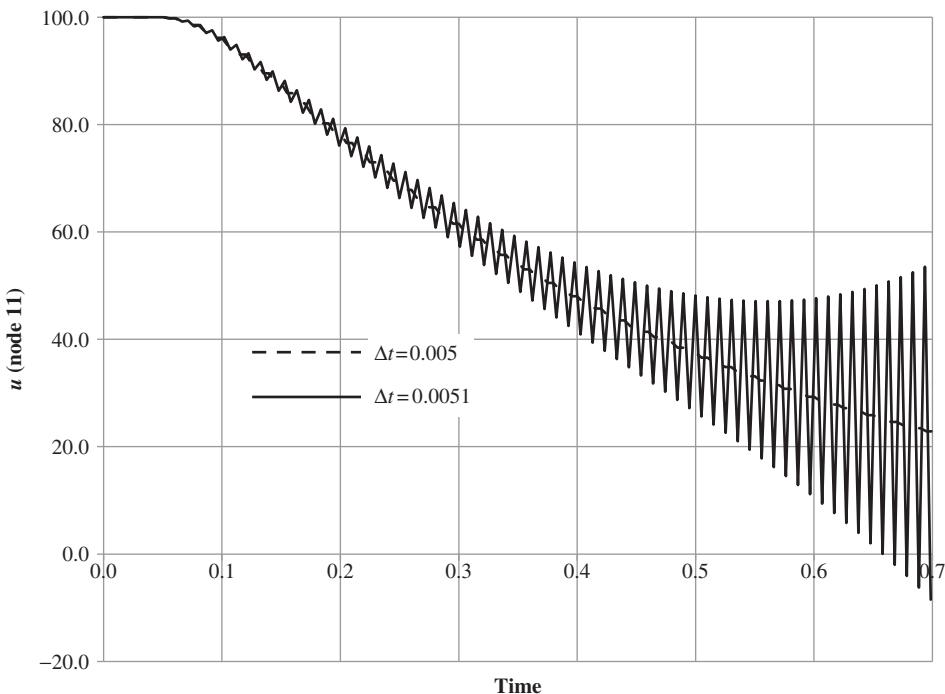
```

This program is for the analysis of 2D ( $\text{ndim}=2$ ) first-order transient problems under plane or axisymmetric conditions, and is closely based on Program 7.2 in Chapter 7. The program includes graphics subroutines `mesh` and `contour` which generate PostScript files containing, respectively, images of the finite element mesh (held in `*.msh`) and a contour map of the excess pore pressure (held in `*.con`) corresponding to the solution at the requested time step `ntime`. The contouring only works currently for meshes made up of 4-node quadrilateral elements.

There are 11 equations

Time	Uav	Pressure (node 11)
0.0000E+00	0.0000E+00	0.1000E+03
0.1000E+00	0.3577E+00	0.9491E+02
0.2000E+00	0.5047E+00	0.7728E+02
0.3000E+00	0.6136E+00	0.6074E+02
0.4000E+00	0.6981E+00	0.4751E+02
0.5000E+00	0.7641E+00	0.3713E+02
0.6000E+00	0.8156E+00	0.2902E+02
0.7000E+00	0.8559E+00	0.2268E+02
0.8000E+00	0.8874E+00	0.1772E+02
0.9000E+00	0.9120E+00	0.1385E+02
0.1000E+01	0.9312E+00	0.1082E+02
0.1100E+01	0.9463E+00	0.8459E+01
0.1200E+01	0.9580E+00	0.6611E+01
0.1300E+01	0.9672E+00	0.5166E+01
0.1400E+01	0.9743E+00	0.4037E+01
0.1500E+01	0.9800E+00	0.3155E+01
0.1600E+01	0.9843E+00	0.2466E+01
0.1700E+01	0.9878E+00	0.1927E+01
0.1800E+01	0.9904E+00	0.1506E+01
0.1900E+01	0.9925E+00	0.1177E+01
0.2000E+01	0.9942E+00	0.9198E+00
Depth Pressure (time=0.1000E+01)		
0.0000E+00	0.0000E+00	
0.1000E+00	0.1693E+01	
0.2000E+00	0.3345E+01	
0.3000E+00	0.4914E+01	
0.4000E+00	0.6362E+01	
0.5000E+00	0.7654E+01	
0.6000E+00	0.8757E+01	
0.7000E+00	0.9644E+01	
0.8000E+00	0.1029E+02	
0.9000E+00	0.1069E+02	
0.1000E+01	0.1082E+02	

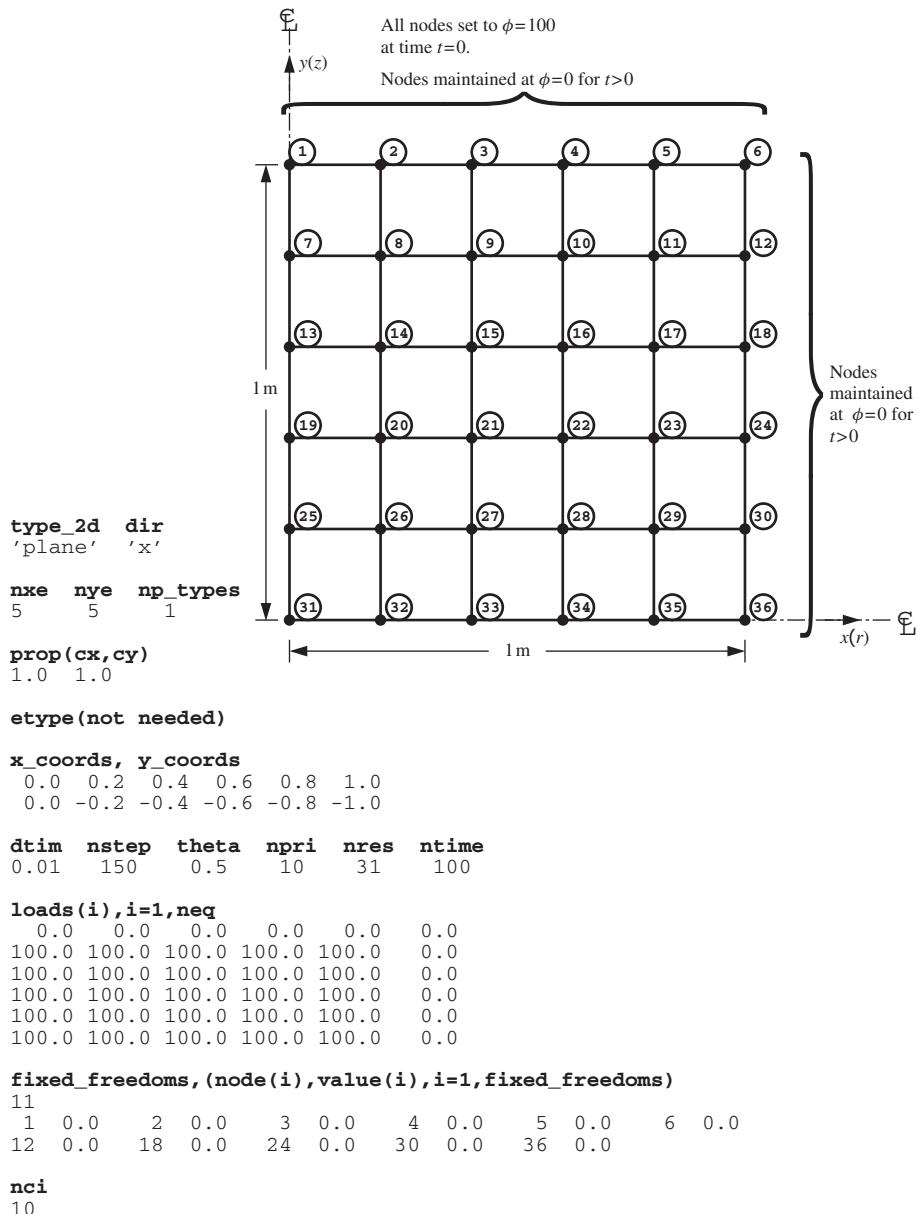
**Figure 8.8** Results from Program 8.3 example



**Figure 8.9** Numerical instability in explicit Program 8.3

The first example chosen is shown in Figure 8.10 and could represent dissipation of excess porewater pressure from a square block of soil with drainage permitted at the outside boundaries. Due to fourfold symmetry, only one-quarter of the square is modelled with excess pore pressure at the outer drained boundaries fixed to zero, and the two inner boundaries defaulting to ‘no-flow’ boundary conditions.

The first line of data reads `type_2d` and `dir` and indicates that a plane analysis is to be performed with element and node numbering in the  $x$ -direction. The second line shows that the rectangular mesh consists of five columns (`nxe`) and five rows (`nye`) of elements, and there is only one property type (`np_types`) in this homogeneous example. The third line reads the 2D coefficients of consolidation  $c_x$  and  $c_y$  into the property array `prop`, and since there is only one property type in this problem, the `etype` data is not required. The fourth and fifth lines give, respectively, the  $x$ - (`x_coords`) and  $y$ -coordinates (`y_coords`) of the lines that make up the mesh. The sixth line of data reads the time-stepping and output parameters with the same meaning as in the data for Program 8.1. The next data read into `loads` is the initial excess pore pressure at all the nodes in the mesh. In this example, the square block is subjected to an initial uniform excess pore pressure of 100.0, with drainage immediately effective at the top and right boundaries. The next line of data indicates that there are 11 (`fixed Freedoms`) nodes to have fixed values. The next two lines of data indicate the node numbers (`node`) and the values (`value`) to which they are to be fixed (zero in the case of drainage boundaries). The final line of data reads `nci=10`, indicating that the contour map of



**Figure 8.10** Mesh and data for the first Program 8.4 and Program 8.7 example

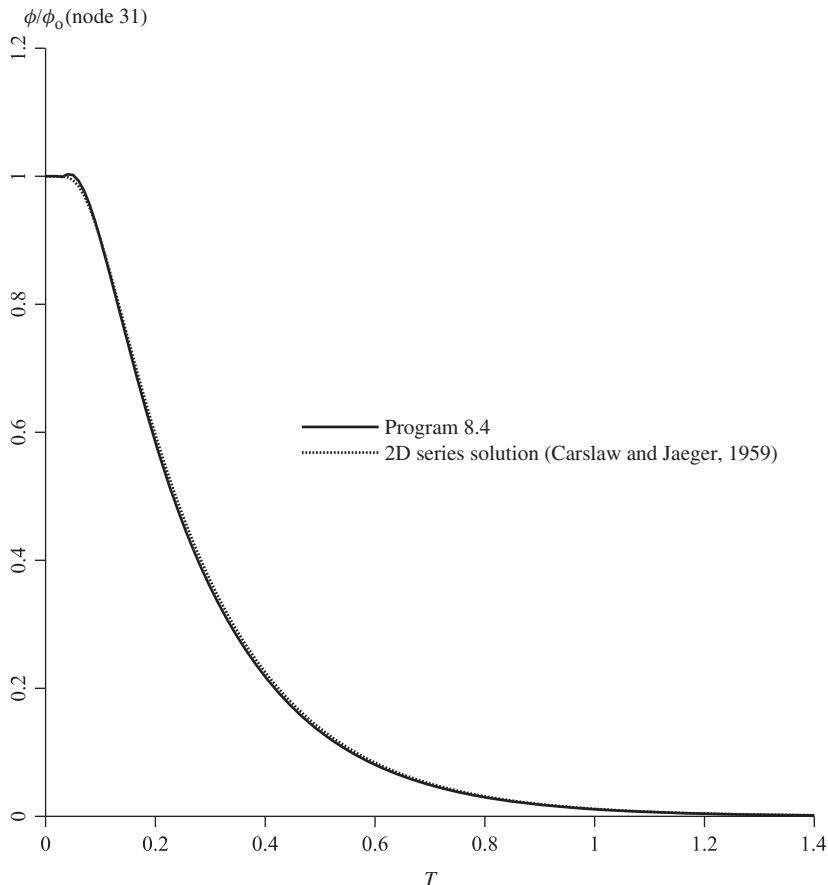
excess pore pressure, corresponding to the situation after  $n_{time}$  time steps and written to file  $*.con$ , will include 10 contour intervals.

The output shown in Figure 8.11 gives the excess pore pressure at node  $n_{res}=31$ , which is the centre of the mesh, every  $n_{pri}=10$  time steps (every 0.1 s). The normalised result at node 31 (after division by the initial value) is plotted against the time factor  $T$  in Figure 8.12, where it is compared with series solution values (e.g., Carslaw

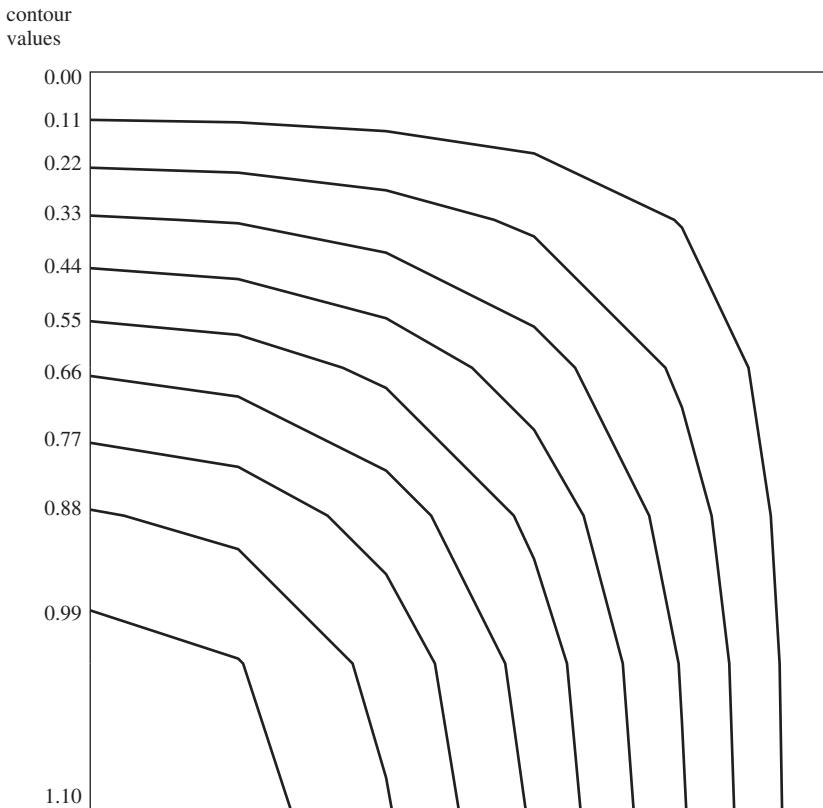
There are 36 equations and the skyline storage is 246

Time	Pressure (node 31)
0.0000E+00	0.1000E+03
0.1000E+00	0.9009E+02
0.2000E+00	0.5845E+02
0.3000E+00	0.3580E+02
0.4000E+00	0.2178E+02
0.5000E+00	0.1324E+02
0.6000E+00	0.8051E+01
0.7000E+00	0.4895E+01
0.8000E+00	0.2976E+01
0.9000E+00	0.1809E+01
0.1000E+01	0.1100E+01
0.1100E+01	0.6687E+00
0.1200E+01	0.4065E+00
0.1300E+01	0.2472E+00
0.1400E+01	0.1503E+00
0.1500E+01	0.9135E-01

**Figure 8.11** Results from first Program 8.4 example



**Figure 8.12** Comparison of Program 8.4 result for a planar region with 2D series solution



**Figure 8.13** Contour map of excess pore pressure after  $t = 1$  from first Program 8.4 example

and Jaeger, 1959). The coarse finite element idealisation gives excellent agreement. The contour map corresponding to the distribution of excess pore pressure after `ntime` time steps ( $t = 1.0$  s) is given in Figure 8.13.

The second example is of the same problem considered previously, but under axisymmetric conditions. In soil mechanics, the physical analogue would be a ‘triaxial’ specimen of soil draining from all its boundaries.

The data shown in Figure 8.14 is very similar to that of the previous example, but with `type_2d` set to ‘axisymmetric’ and `dir` set to ‘r’, because the node and element numbering is now in the radial direction. The number of integrating points `nip` remains equal to 4 for the rectangular 4-node elements considered in this example (see discussion of `nip` in Program 7.2). The output at node 31, shown in Figure 8.15, is plotted in Figure 8.16 and compared again with Carslaw and Jaeger’s (1959) axisymmetric solution.

```

type_2d dir
'axisymmetric' 'r'

nxe nyx np_types
5      5      1

prop(cx,cy)
1.0  1.0

etype(not needed)

x_coords, y_coords
0.0  0.2  0.4  0.6  0.8  1.0
0.0 -0.2 -0.4 -0.6 -0.8 -1.0

dtime nstep theta npri nres ntime
0.01  150    0.5   10    31   100

loads(i),i=1,neq
0.0  0.0  0.0  0.0  0.0  0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0

fixed Freedoms, (node(i),value(i),i=1,fixed Freedoms)
11
1  0.0   2  0.0   3  0.0   4  0.0   5  0.0   6  0.0
12 0.0   18 0.0   24 0.0   30 0.0   36 0.0

nci
10

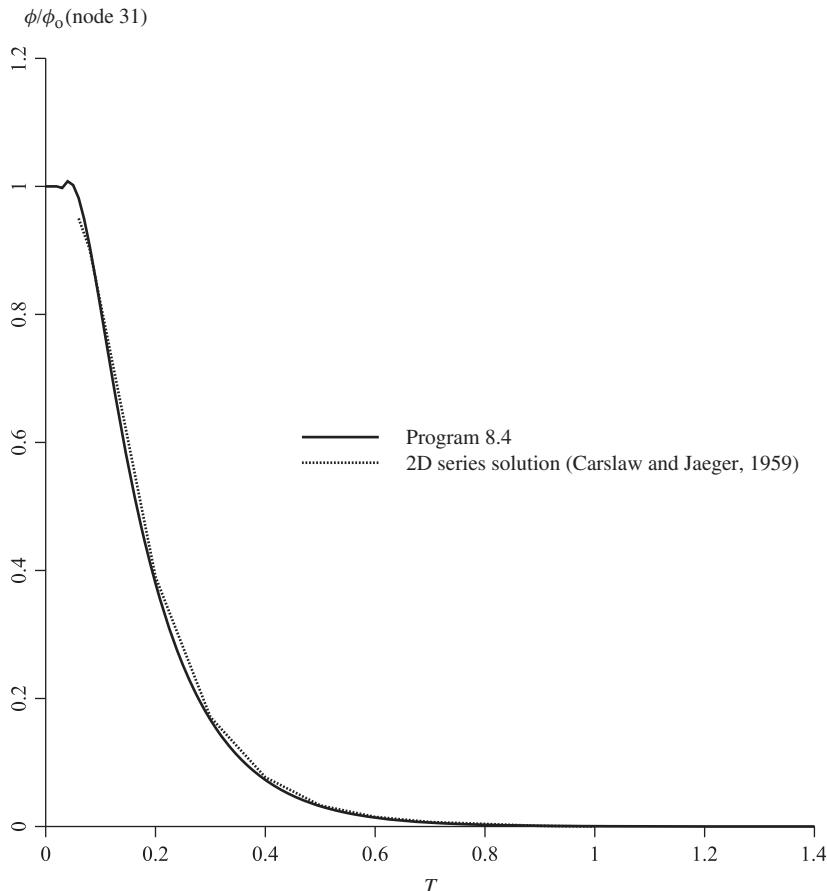
```

**Figure 8.14** Data for second Program 8.4 example

There are 36 equations and the skyline storage is 246

Time	Pressure (node 31)
0.0000E+00	0.1000E+03
0.1000E+00	0.8096E+02
0.2000E+00	0.3798E+02
0.3000E+00	0.1667E+02
0.4000E+00	0.7280E+01
0.5000E+00	0.3176E+01
0.6000E+00	0.1386E+01
0.7000E+00	0.6046E+00
0.8000E+00	0.2638E+00
0.9000E+00	0.1151E+00
0.1000E+01	0.5022E-01
0.1100E+01	0.2191E-01
0.1200E+01	0.9560E-02
0.1300E+01	0.4171E-02
0.1400E+01	0.1820E-02
0.1500E+01	0.7940E-03

**Figure 8.15** Results from second Program 8.4 example



**Figure 8.16** Comparison of Program 8.4 result for a cylindrical region with 2D series solution

**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**

```
PROGRAM p85
!-----
! Program 8.5 Plane or axisymmetric consolidation analysis using 4-node
!           rectangular quadrilaterals. Mesh numbered in x(r)- or y(z)-
!           direction. Implicit time integration using the "theta"
!           method. No global matrix assembly. Diagonally preconditioned
!           conjugate gradient solver
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,fixed Freedoms,i,iel,j,k,nci,ndim=2,nels,neq, &
nip=4,nlen,nn,nod=4,npri,np_types,nres,nstep,ntime,nxe,nye
REAL(iwp)::alpha,beta,cg_tol,det,dtim,one=1.0_iwp,penalty=1.0e20_iwp, &
```

```

theta,time,up,zero=0.0_iwp; LOGICAL::cg_converged
CHARACTER(LEN=15)::argv,dir,element='quadrilateral',type_2d
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:, ),g_num(:, :, ),node(:, ),num(:, )
REAL(iwp),ALLOCATABLE::coord(:, :, ),d(:, ),der(:, :, ),deriv(:, :, )
diag_precon(:, ),fun(:, ),gc(:, ),g_coord(:, :, ),jac(:, :, ),kay(:, :, ),kc(:, :, )
&
loads(:, ),ntn(:, :, ),p(:, ),mm(:, :, ),points(:, :, ),prop(:, :, ),r(:, ),store(:, ),
&
storka(:, :, :, ),storkb(:, :, :, ),u(:, ),value(:, ),weights(:, ),x(:, ),xnew(:, ),
&
x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,dir,nxe,nye,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
WRITE(11,'(a,i5,a)')" There are",neq," equations"
ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim), &
fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),&
mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod), &
storka(nod,nod,nels),storkb(nod,nod,nels),etype(nels),x_coords(nxe+1), &
Y_coords(nye+1),prop(ndim,np_types),loads(0:neq),diag_precon(0:neq), &
u(0:neq),d(0:neq),p(0:neq),x(0:neq),r(0:neq),xnew(0:neq),gc(ndim))
READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres,ntime
!-----loop the elements to set up element data-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
CALL sample(element,points,weights); diag_precon=zero; gc=one
!-----element matrix integration, storage and preconditioner-----
elements_2: DO iel=1,nels
    key=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); IF(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
        CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
    END DO gauss_pts
    storka(:, :, iel)=mm+kc*theta*dtim; storkb(:, :, iel)=mm-kc*(one-theta)*dtim
    DO k=1,nod
        diag_precon(num(k))=diag_precon(num(k))+storka(k,k,iel)
    END DO
END DO elements_2
!-----specify initial and boundary values-----
READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms), &
    store(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    diag_precon(node)=diag_precon(node)+penalty; store=diag_precon(node)
END IF
diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero
!-----time stepping loop-----
WRITE(11,'(/a,i3,a)')" Time      Pressure (node",nres,")  cg iters"
WRITE(11,'(2e12.4)')0.0,loads(nres)
timesteps: DO j=1,nstep; time=j*dtim; u=zero
    elements_3 : DO iel=1,nels
        num=g_num(:,iel); kc=storkb(:, :, iel)

```

```

        u(num)=u(num)+MATMUL(kc,loads(num))
    END DO elements_3
    u(0)=zero; r=u; IF(fixed_freedoms/=0)r(node)=store*value
    d=diag_precon*r; p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
    pcg: DO; cg_iters=cg_iters+1; u=zero
    elements_4: DO iel=1,nels
        num=g_num(:,iel); kc=storka(:,:,iel)
        u(num)=u(num)+MATMUL(kc,p(num))
    END DO elements_4
    IF(fixed_freedoms==0)u(node)=p(node)*store; up=DOT_PRODUCT(r,d)
    alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha; r=r-u*alpha
    d=diag_precon*r; beta=DOT_PRODUCT(r,d)/up; p=d+p*beta
    CALL checon(xnew,x,cg_tol,cg_converged)
    IF(cg_converged.OR.cg_iters==cg_limit)EXIT
    END DO pcg; loads=xnew
    IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
        CALL contour(loads,g_coord,g_num,nci,argv,nlen,13); END IF
    IF(j/npri*npri==j)
        WRITE(11,'(2e12.4,7x,i5)')time,loads(nres),cg_iters
    END DO timesteps
STOP
END PROGRAM p85

```

The ‘implicit’ Programs 8.1, 8.2 and 8.4 were seen to consist of a linear equation solution on every time step. In this program, this solution is accomplished iteratively using pcg as was done in previous chapters.

In this program, all the element conductivity and ‘mass’ matrices are stored in the forms ( $[\mathbf{m}_m] + \theta \Delta t [\mathbf{k}_c]$ ) and ( $[\mathbf{m}_m] - (1 - \theta) \Delta t [\mathbf{k}_c]$ ) in arrays storka and storkb, respectively, as required by equation (3.100). The diagonal preconditioner is formed from the diagonal terms of the first of these as it would have been assembled. The only additional inputs compared to Program 8.4 are the iteration tolerance, cg\_tol, and the limiting number of pcg iterations, cg\_limit. The data are shown as Figure 8.17, with output as Figure 8.18. For an iteration tolerance of 0.0001 and the same time step as was used in Program 8.4, the pcg process converges in at most three iterations, and leads to the same solution as given in Figure 8.11.

## 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

```

PROGRAM p86
!-----
! Program 8.6 Plane or axisymmetric analysis of the consolidation equation
!           using 4-node rectangular quadrilaterals. Mesh numbered in
!           x(r)- or y(z)- direction. Explicit time integration using
!           the "theta=0" method.
!
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,j,nci,ndim=2,nels,neq,nip=4,nlen,nn,nod=4, &
npri,np_types,nres,nstep,ntime,nxe,nye
REAL(iwp)::det,dtim,one=1.0_iwp,time,zero=0.0_iwp
CHARACTER(len=15)::argv,dir,element='quadrilateral',type_2d
!-----dynamic arrays-----

```

```

INTEGER,ALLOCATABLE::etype(:,g_num(:,:,),node(:,),num(:)
REAL(iwp),ALLOCATABLE::coord(:, :,),der(:, :,),deriv(:, :,),fun(:, ),gc(:, ),
globma(:, ),g_coord(:, :,),jac(:, :,),kay(:, :,),kc(:, :,),loads(:, ),mass(:, ),
newlo(:, ),ntn(:, :,),mm(:, :,),points(:, :,),prop(:, :,),store_mm(:, :, :, ),
value(:, ),weights(:, ),x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,dir,nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
WRITE(11,'(a,i5,a)')" There are",neq," equations"
ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),globma(0:neq),
coord(nod,ndim),fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),&
deriv(ndim,nod),mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),&
num(nod),etype(nels),x_coords(nxe+1),y_coords(nye+1),&
prop(ndim,np_types),gc(ndim),store_mm(nod,nod,nels),mass(nod),&
loads(0:neq),newlo(0:neq))
READ(10,*)prop; etype=1; IF(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,npri,nres,ntime
CALL sample(element,points,weights); globma=zero; gc=one
!-----create and store element and global lumped mass matrices-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); if(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
        CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
    END DO gauss_pts
    DO i=1,nod; mass(i)=SUM(mm(i,:)); END DO; mm=zero
    DO i=1,nod; mm(i,i)=mass(i); END DO
    store_mm(:, :, iel)=mm-kc*dtim; globma(num)=globma(num)+mass
END DO elements_1; globma(1:)=one/globma(1:)
CALL mesh(g_coord,g_num,argv,nlen,12)
!-----specify initial and boundary values-----
READ(10,*)loads(1:); loads(0)=zero; READ(10,*)fixed_freedoms
IF(fixed_freedoms/=0)then
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
END IF
!-----time stepping loop-----
WRITE(11,'(/a,i3,a)')" Time Pressure (node",nres,")"
WRITE(11,'(2e12.4)')0.0,loads(nres)
timesteps: DO j=1,nstep; time=j*dtim; newlo=zero
    elements_2: DO iel=1,nels
        num=g_num(:,iel); mm=store_mm(:, :, iel)
        newlo(num)=newlo(num)+MATMUL(mm,loads(num))
    END DO elements_2; newlo(0)=zero; loads=newlo*globma
    IF(fixed_freedoms/=0)loads(node)=value
    IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
        CALL contour(loads,g_coord,g_num,nci,argv,nlen,13); END IF
        IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
    END DO timesteps
STOP
END PROGRAM p86

```

```

type_2d dir
'plane' 'x'

nxe nye cg_tol cg_limit np_types
5      5     0.0001    100      1

prop(cx,cy)
1.0   1.0

etype(not needed)

x_coords, y_coords
0.0  0.2  0.4  0.6  0.8  1.0
0.0 -0.2 -0.4 -0.6 -0.8 -1.0

dtime nstep theta npri nres ntime
0.01 150    0.5    10    31    100

loads(i), i=1, neq
0.0  0.0  0.0  0.0  0.0  0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0

fixed_freedoms, (node(i), value(i), i=1, fixed_freedoms)
11
1  0.0   2  0.0   3  0.0   4  0.0   5  0.0   6  0.0
12 0.0   18 0.0   24 0.0   30 0.0   36 0.0

nci
10

```

**Figure 8.17** Data for Program 8.5 example

There are 36 equations

Time	Pressure (node 31)	cg iters
0.0000E+00	0.1000E+03	
0.1000E+00	0.9009E+02	3
0.2000E+00	0.5845E+02	3
0.3000E+00	0.3580E+02	2
0.4000E+00	0.2178E+02	2
0.5000E+00	0.1324E+02	2
0.6000E+00	0.8051E+01	2
0.7000E+00	0.4895E+01	2
0.8000E+00	0.2976E+01	2
0.9000E+00	0.1809E+01	2
0.1000E+01	0.1100E+01	2
0.1100E+01	0.6687E+00	2
0.1200E+01	0.4065E+00	2
0.1300E+01	0.2472E+00	2
0.1400E+01	0.1503E+00	2
0.1500E+01	0.9135E-01	2

**Figure 8.18** Results from Program 8.5 example

This 2D ‘explicit’ program uses the same algorithm described for Program 8.3 and the structure chart shown in Figure 3.22.

The problem shown in Figure 8.10 has been analysed again with the data shown in Figure 8.19, with output as Figure 8.20. The only difference from Figure 8.10 is that being an ‘explicit’ algorithm, theta is automatically set to zero and hence it is no longer required as data.

```

type_2d dir
'plane' 'x'

nxe nye np_types
5      5      1

prop(cx,cy)
1.0  1.0

etype(not needed)

x_coords, y_coords
0.0  0.2  0.4  0.6  0.8  1.0
0.0 -0.2 -0.4 -0.6 -0.8 -1.0

dtime nstep npri nres ntime
0.01   150    10     31     100

loads(i),i=1,neg
0.0  0.0  0.0  0.0  0.0  0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0
100.0 100.0 100.0 100.0 100.0 0.0

fixed_freedoms,(node(i),value(i),i=1,fixed_freedoms)
11
1  0.0  2  0.0  3  0.0  4  0.0  5  0.0  6  0.0
12 0.0  18 0.0  24 0.0  30 0.0  36 0.0

nci
10

```

**Figure 8.19** Data for Program 8.6 example

There are 36 equations

Time	Pressure (node 31)
0.0000E+00	0.1000E+03
0.1000E+00	0.8972E+02
0.2000E+00	0.5881E+02
0.3000E+00	0.3624E+02
0.4000E+00	0.2215E+02
0.5000E+00	0.1353E+02
0.6000E+00	0.8258E+01
0.7000E+00	0.5042E+01
0.8000E+00	0.3078E+01
0.9000E+00	0.1879E+01
0.1000E+01	0.1147E+01
0.1100E+01	0.7005E+00
0.1200E+01	0.4277E+00
0.1300E+01	0.2611E+00
0.1400E+01	0.1594E+00
0.1500E+01	0.9733E-01

**Figure 8.20** Results from Program 8.6 example

## 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

```

PROGRAM p87
!-----
! Program 8.7 Plane or axisymmetric analysis of the consolidation equation
!           using 4-node rectangular quadrilaterals. Mesh numbered in
!           x(r)- or y(z)- direction. "theta" method using an
!           "element-by-element" (ebe) product algorithm.
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,j,nci,ndim=2,nels,neq,nip=4,nlen,nn,nod=4, &
npri,np_types,nres,nstep,ntime,nxe,nye
REAL(iwp)::det,dtim,one=1.0_iwp,pt5=0.5_iwp,theta,time,zero=0.0_iwp
CHARACTER(LEN=15)::argv,dir,element='quadrilateral',type_2d
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g_num(:,,:),node(:,),num(:)
REAL(iwp),ALLOCATABLE::coord(:,,:),der(:,,:),deriv(:,,:),fun(:,),gc(:,),
globma(:,),g_coord(:,,:),jac(:,,:),kay(:,,:),kc(:,,:),loads(:,),ntn(:,,:), &
mm(:,,:),points(:,,:),prop(:,,:),store_kc(:,,:,:),value(:,),weights(:,),
x_coords(:,),y_coords(:,))

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,dir,nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
WRITE(11,'(a,i5,a)')" There are",neq," equations"
ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim), &
fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),&
mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod), &
globma(0:nn),store_kc(nod,nod,nels),gc(ndim),loads(0:neq), &
x_coords(nxe+1),y_coords(nye+1),prop(ndim,np_types),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres,ntime
CALL sample(element,points,weights); globma=zero; gc=one
!-----create and store element and global lumped mass matrices-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); if(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
        CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)*gc(1)
    END DO gauss_pts; store_kc(:, :,iel)=kc
    DO i=1,nod; globma(num(i))=globma(num(i))+SUM(mm(i,:)); END DO
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
!-----recover element a and b matrices-----
elements_2: DO iel=1,nels
    num=g_num(:,iel); kc=-store_kc(:, :,iel)*(one-theta)*dtim*pt5
    mm=store_kc(:, :,iel)*theta*dtim*pt5
    DO i=1,nod

```

```

mm(i,i)=mm(i,i)+globma(num(i)); kc(i,i)=kc(i,i)+globma(num(i))
END DO; CALL invert(mm); mm=MATMUL(mm,kc); store_kc(:,:,iel)=mm
END DO elements_2
!-----specify initial and boundary values-----
READ(10,*) loads(1:); loads(0)=zero; READ(10,*) fixed_freedoms
IF(fixed_freedoms/=0)THEN
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
END IF
!-----time stepping loop-----
WRITE(11,'(/a,i3,a)')      Time      Pressure (node",nres,"")
WRITE(11,'(2e12.4)')0.0,loads(nres)
timesteps: DO j=1,nstep
    time=j*dtim
!-----first pass (1 to nels)-----
elements_3: DO iel=1,nels
    num=g_num(:,iel); mm=store_kc(:,:,iel)
    loads(num)=MATMUL(mm,loads(num)); loads(0)=zero; loads(node)=value
END DO elements_3
!-----second pass (nels to 1)-----
elements_4: DO iel=nels,1,-1
    num=g_num(:,iel); mm=store_kc(:,:,iel)
    loads(num)=MATMUL(mm,loads(num)); loads(0)=zero; loads(node)=value
END DO elements_4
IF(nod==4.AND.j==ntime)THEN; READ(10,*)nci
    CALL contour(loads,g_coord,g_num,nci,argv,nlen,13); END IF
    IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
END DO timesteps
STOP
END PROGRAM p87

```

The motivation in using this algorithm is to preserve the storage economy achieved by the previous explicit technique while attaining the stability properties enjoyed by implicit methods typified by Programs 8.1, 8.2 and 8.4 without involving solution of sets of global equations. The process is described in Chapter 3 by equations (3.105)–(3.108) and in structure by Figure 3.23. The program bears a resemblance to Program 8.6. The element integration loop is employed to store the element matrices  $[\mathbf{k}_c]$  in a storage array `store_kc`. In addition, the element consistent mass matrices held in `mm` are diagonalised and the global mass vector, `globma`, assembled. A second loop over the elements is then made, headed ‘recover element matrices’. These are the matrices given in Figure 3.23 by  $[\mathbf{m}_m] - (1 - \theta)\Delta t[\mathbf{k}_c]/2$  and  $[\mathbf{m}_m] + \theta\Delta t[\mathbf{k}_c]/2$ , and are called `kc` and `mm`, respectively in the program. The algorithm calls for  $[\mathbf{b}]$  (`mm`) to be inverted, which is done using the library subroutine `invert`. Then  $[\mathbf{a}]$  is formed as  $[\mathbf{b}]^{-1}[[\mathbf{m}_m] - (1 - \theta)\Delta t[\mathbf{k}_c]/2]$ , by multiplying `mm` and `kc`. The result, called `mm` in the program, is re-stored as `store_kc`.

Initial conditions can then be prescribed and the time-stepping loop entered. Within that loop, two passes are made over the elements from first to last and back again. Half of the total  $\Delta t[\mathbf{k}_c]$  increment operates on each pass, and this has been accounted for already in forming  $[\mathbf{a}]$  and  $[\mathbf{b}]$ . The essential coding recovers each element  $[\mathbf{b}]^{-1}[\mathbf{a}]$  matrix from `store_kc` and multiplies it by the appropriate part of the solution `loads`. Note that in this product algorithm the solution is continually being updated so there is no need for any ‘new loads’ vector such as had to be employed in the explicit summation algorithm.

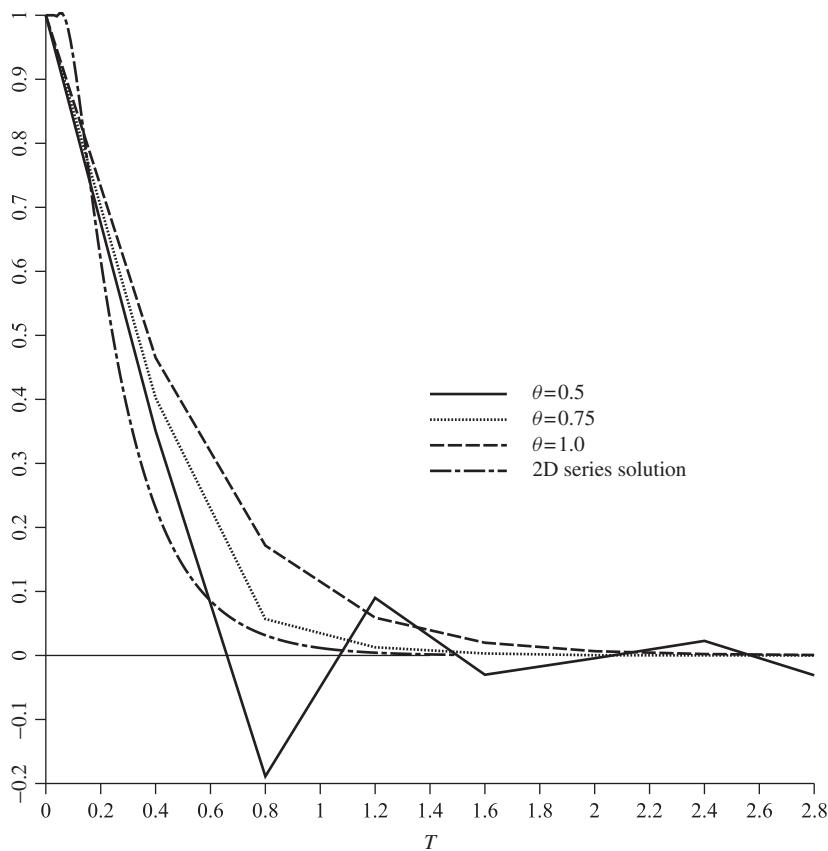
The data for this example are identical to those given in Figure 8.10, with results given in Figure 8.21.

There are 36 equations

Time	Pressure (node 31)
0.0000E+00	0.1000E+03
0.1000E+00	0.8912E+02
0.2000E+00	0.6107E+02
0.3000E+00	0.3894E+02
0.4000E+00	0.2450E+02
0.5000E+00	0.1537E+02
0.6000E+00	0.9644E+01
0.7000E+00	0.6050E+01
0.8000E+00	0.3795E+01
0.9000E+00	0.2380E+01
0.1000E+01	0.1493E+01
0.1100E+01	0.9366E+00
0.1200E+01	0.5875E+00
0.1300E+01	0.3685E+00
0.1400E+01	0.2312E+00
0.1500E+01	0.1450E+00

**Figure 8.21** Results from Program 8.7 example

$\phi/\phi_0$ (node 31)



**Figure 8.22** Typical solutions from Program 8.4 with varying  $\theta$  ( $\Delta t = 0.4$ )

## 8.2 Comparison of Programs 8.4, 8.5, 8.6 and 8.7

These four programs can all be used to solve plane or axisymmetric transient heat conduction or uncoupled soil consolidation problems. Comparison of Figures 8.11, 8.18, 8.20 and 8.21 shows that for the chosen problem—at the time step used (that is 0.01)—all solutions are accurate, and indeed the explicit solution (Figure 8.20) is probably as accurate as any despite being the simplest and cheapest to obtain.

It must, however, be remembered that as demonstrated in Figure 8.9 for 1D analysis, as the time step  $\Delta t$  is increased, the explicit algorithm will lead to unstable results (the stability limit for the selected 2D problem is about  $\Delta t_{crit} = 0.02$ ). For time steps larger than the critical value, implicit algorithms such as in Program 8.4 with  $\theta = 0.5$  will tend to produce oscillatory results, which can be damped, at the expense of average accuracy, by increasing  $\theta$  towards 1.0. Typical behaviour of the implicit algorithm is illustrated in Figure 8.22.

Program 8.7, while retaining the storage economies of Program 8.6, allows the time step to be increased well beyond the explicit limit. For example, in the selected problem, reasonable results are still produced at  $\Delta t = 10\Delta t_{crit}$ . However, as  $\Delta t$  is increased still further, accuracy becomes poorer and Program 8.4 yields the best solutions for very large  $\Delta t$ .

It will be clear that algorithm choice in this area is not a simple one and depends on the nature of the problem (degree of non-linearity, etc.) and on the hardware employed. ‘Element-by-element’ methods afford much scope for parallelisation and this is exploited in Chapter 12.

### Program 8.8 General two- (plane) or three-dimensional transient (consolidation) analysis. Implicit time integration using the ‘theta’ method

```
PROGRAM p88
!-----
! Program 8.8 General two- (plane) or three-dimensional analysis of the
! consolidation equation. Implicit time integration using
! the "theta" method.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,j,nci,ndim,nels,neq,nip,nlen,nn,nod=4,npri,&
np_types,nres,nstep,ntime; CHARACTER(len=15)::argv,element
REAL(iwp)::det,dtim,penalty=1.0e20_iwp,theta,time,zero=0.0_iwp
!----- dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:, :, :),g_num(:, :, :),node(:, :, :),num(:, :, :),kdiag(:, :, :),
REAL(iwp),ALLOCATABLE::bp(:, :, :),coord(:, :, :),der(:, :, :),deriv(:, :, :),fun(:, :, :),      &
g_coord(:, :, :),jac(:, :, :),kay(:, :, :),kc(:, :, :),kv(:, :, :),loads(:, :, :),newlo(:, :, :),      &
ntn(:, :, :),mm(:, :, :),points(:, :, :),prop(:, :, :),storbp(:, :, :),value(:, :, :),weights(:, :, :))
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)element,nels,nn,nip,nod,ndim,np_types; neq=nn
ALLOCATE(points(nip,ndim),g_coord(ndim,nn),coord(nod,ndim),fun(nod),      &
etype(nels),jac(ndim,ndim),weights(nip),num(nod),ntn(nod,nod),      &
g_num(nod,nels),der(ndim,nod),deriv(ndim,nod),kc(nod,nod),mm(nod,nod),      &
kay(ndim,ndim),kdiag(neq),prop(ndim,np_types),newlo(0:neq),loads(0:neq))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)g_coord; READ(10,*)g_num; READ(10,*)dtim,nstep,theta,npri,nres
IF(ndim==2.AND.nod==4)THEN
  READ(10,*)ntime,nci; CALL mesh(g_coord,g_num,argv,nlen,12)
END IF; kdiag=0
```

```

!-----loop the elements to set up global geometry and kdiag -----
elements_1: DO iel=1,nels
    num=g_num(:,iel); CALL fkdiag(kdiag,num)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
WRITE(11,'(2(a,i5))') &
" There are",neq," equations and the skyline storage is ",kdiag(neq)
ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq)))
CALL sample(element,points,weights); kv=zero; bp=zero
!----- global conductivity matrix assembly-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
        CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)
    END DO gauss_pts
    CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
END DO elements_2; kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta
!-----initial and boundary conditions data-----
READ(10,*) loads(1:); loads(0)=zero; READ(10,*) fixed_freedoms
IF(fixed_freedoms/=0)then
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms), &
              storbp(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
END IF
!-----factorise left hand side-----
CALL sparin(bp,kdiag)
!-----time stepping recursion-----
WRITE(11,'(/a,i3,a)') " Time Pressure (node",nres,")"
WRITE(11,'(2e12.4)') 0.0,loads(nres)
timesteps: DO j=1,nstep
    time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag)
    IF(fixed_freedoms/=0) newlo(node)=storbp*value
    CALL spabac(bp,newlo,kdiag); loads=newlo
    IF(ndim==2.AND.nod==4.AND.j==ntime) &
        CALL contour(loads,g_coord,g_num,nci,argv,nlen,13)
    IF(j/npri*npri==j) WRITE(11,'(2e12.4)') time,loads(nres)
END DO timesteps
STOP
END PROGRAM p88

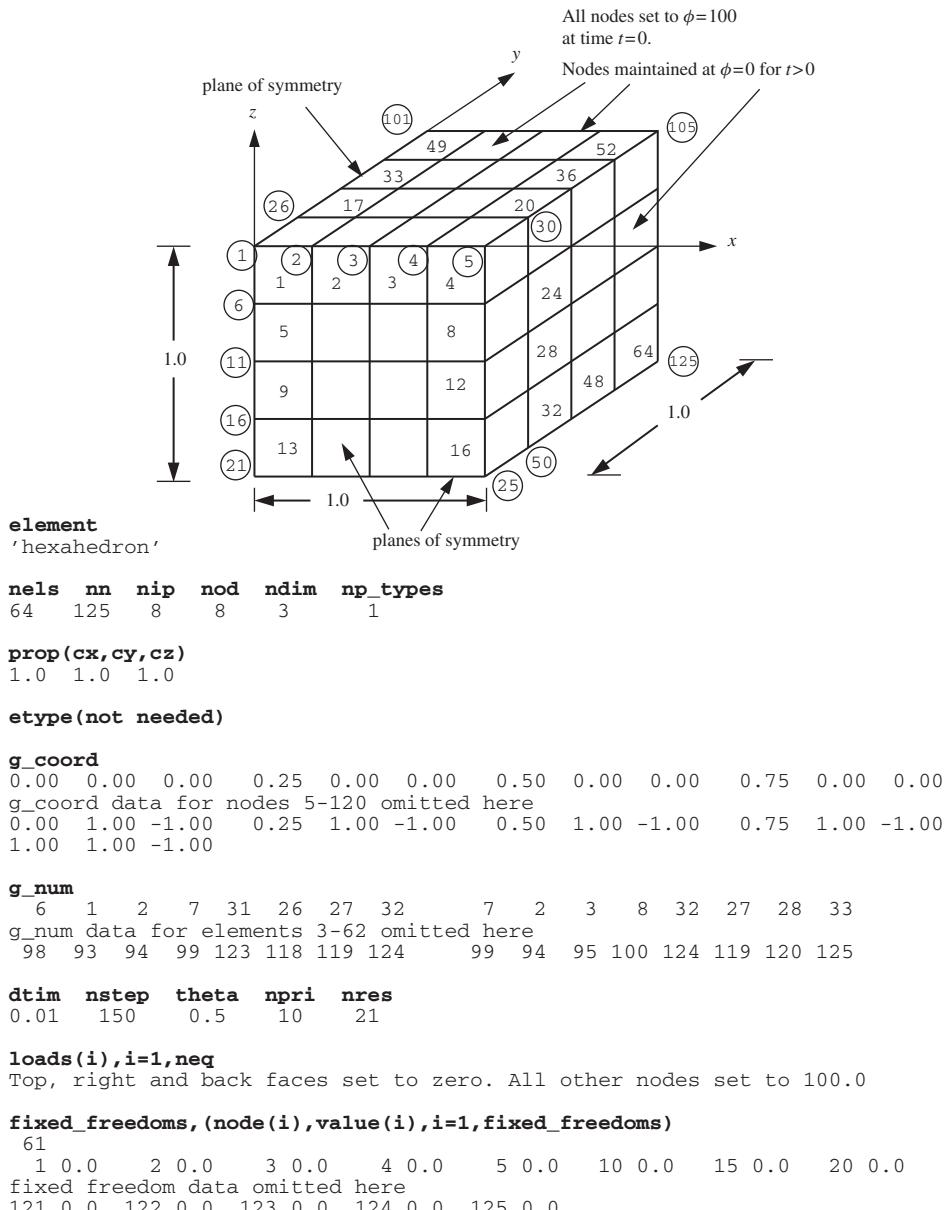
```

Program 8.8 is the last of the diffusion or ‘consolidation’ programs in this chapter and the most general. The program can analyse consolidation over any two- or three-dimensional domain with non-homogeneous and anisotropic material properties. The program is very similar to Program 5.4 for general elastic analysis and Program 7.4 for general steady seepage analysis. As with those programs, a variety of 2- or 3D elements can be chosen through the data file. The graphics subroutines mesh and contour are only activated for 2D analysis, and in the current versions, contouring is only available when using 4-node quadrilateral elements.

The main difference from earlier programs in the chapter (e.g., Program 8.4) is that this program includes no ‘geometry’ subroutine, so all nodal coordinates (*g\_coords*) and element node numbers (*g\_num*) must be provided as data. In addition, some of the

variables that were previously fixed in the declaration statements, must now be read as data in order to identify the dimensionality of the problem and the type of element required. There are no variables required by this program that have not already been encountered in earlier programs of this chapter.

A 3D consolidation example is shown in Figure 8.23. The model represents one-eighth of a symmetrical cube with drainage permitted at all its outer surfaces. The finite element model uses 8-node hexahedral elements, and includes 64 elements and 125 nodes. The



**Figure 8.23** Mesh and data for Program 8.8 example

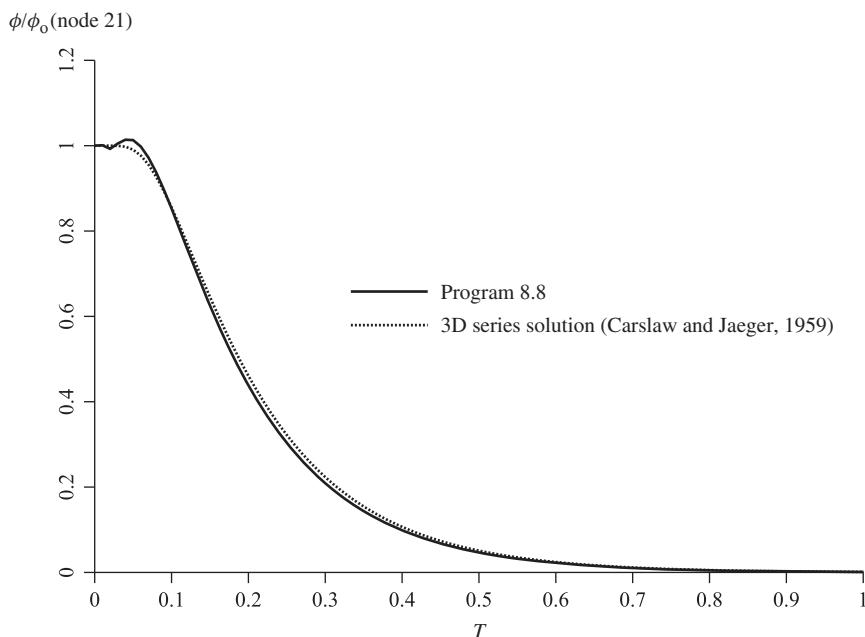
initial condition is that the excess pore pressure at all nodes is set equal to 100.0. The objective is to compute the excess pore pressure at the centre of the cube as time passes.

The first line of data identifies the element type, which in this case is a ‘hexahedron’. The next line gives the number of elements `nels`, the number of nodes `nn`, the number of integrating points `nip`, the number of nodes on each element `nod`, the number of dimensions of the problem `ndim` and the number of property types `np_types`. It may

```
There are 125 equations and the skyline storage is 3225
```

Time	Pressure (node 21)
0.0000E+00	0.1000E+03
0.1000E+00	0.8528E+02
0.2000E+00	0.4386E+02
0.3000E+00	0.2090E+02
0.4000E+00	0.9878E+01
0.5000E+00	0.4666E+01
0.6000E+00	0.2204E+01
0.7000E+00	0.1041E+01
0.8000E+00	0.4916E+00
0.9000E+00	0.2322E+00
0.1000E+01	0.1097E+00
0.1100E+01	0.5179E-01
0.1200E+01	0.2446E-01
0.1300E+01	0.1155E-01
0.1400E+01	0.5457E-02
0.1500E+01	0.2577E-02

**Figure 8.24** Results from Program 8.8 example



**Figure 8.25** Comparison of Program 8.8 result for a cube with the 3D series solution

also be noted that numerical integration of an 8-node hexahedral element usually involves eight Gauss points (two in each of the three coordinate directions), so `nip` is read as 8.

The problem is homogeneous and isotropic, so the next line indicates that the soil has coefficients of consolidation equal to unity in all three coordinate directions ( $c_x = c_y = c_z = 1.0$ ). With `np_types=1`, no `etype` data is required.

The next data involves the 125 ( $x, y, z$ ) coordinates of the mesh read into `g_coord`, followed by the 64 groups of eight node numbers attached to each element read into `g_num`. If dealing with a 3D 8-node element, for example, the order in which the node numbers are read must follow the ordering described in Appendix B. Due to the volume of data required in this example, a truncated version of the data is actually shown in Figure 8.23.

The time-stepping and output data follows next, and involves the usual parameters. Output is requested at node `nres=21`. It should be noted that `ntime` and `nci` are not read in this case because there is no contouring option in 3D.

The output from the program is shown in Figure 8.24, and gives the rate of pore pressure dissipation at the centre of the cube. Figure 8.25 shows a plot of this result compared with the 3D series solution from Carslaw and Jaeger (1959).

## 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

```
PROGRAM p89
!-----
! 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.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::iel,j,nci,ndim=2,nels,neq,nip=4,nlen,nn,nod=4,npri,np_types,  &
nres,nstep,ntime,nxe,nye
REAL(iwp)::det,dtim,d6=6.0_iwp,d12=12.0_iwp,f1,f2,pt25=0.25_iwp,theta,  &
time,two=2.0_iwp,ux,uy,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g_num(:,:,),num(:),kdiag(:)
REAL(iwp),ALLOCATABLE::ans(:),bp(:),coord(:,:,),der(:,:,),deriv(:,:,),
fun(:,:,),g_coord(:,:,),jac(:,:,),kay(:,:,),kc(:,:,),kv(:),loads(:),ntn(:,:,),
mm(:,:,),points(:,:,),prop(:,:,),weights(:,),x_coords(:,),y_coords(:,)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim),  &
fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),  &
mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod),  &
prop(ndim,np_types),x_coords(nxe+1),y_coords(nye+1),etype(nels),  &
kdiag(neq),loads(0:neq),ans(0:neq))
READ(10,*)prop; etype=1; if(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords
```

```

READ(10,*)dtim,nstep,theta,npri,nres,ntime,ux,uy,nci; kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
    CALL fkdiag(kdiag,num)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq)))
WRITE(11,'(2(a,i5),/)')
&
" There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); kv=zero; bp=zero
!-----global conductivity and "mass" matrix assembly-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
        CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)
    END DO gauss_pts
    kc=kc+mm*(ux*ux/kay(1,1)+uy*uy/kay(2,2))*pt25; mm=mm/(theta*dtim)
!-----derivative boundary conditions-----
IF(iel==1)THEN; det=x_coords(2)-x_coords(1)
    kc(2,2)=kc(2,2)+uy*det/d6; kc(2,3)=kc(2,3)+uy*det/d12
    kc(3,2)=kc(3,2)+uy*det/d12; kc(3,3)=kc(3,3)+uy*det/d6
ELSE IF(iel==nels)THEN; det=x_coords(2)-x_coords(1)
    kc(1,1)=kc(1,1)+uy*det/d6; kc(1,4)=kc(1,4)+uy*det/d12
    kc(4,1)=kc(4,1)+uy*det/d12; kc(4,4)=kc(4,4)+uy*det/d6
END IF; CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
END DO elements_2; f1=uy*det/(two*theta); f2=f1; bp=bp+kv; kv=bp-kv/theta
!-----factorise equations-----
CALL sparin(bp,kdiag); loads=zero
!-----time stepping loop-----
WRITE(11,'(a,i3,a)')" Time Concentration (node",nres,")"
WRITE(11,'(2e12.4)')0.0,loads(nres)
timesteps: DO j=1,nstep
    time=j*dtim; CALL linmul_sky(kv,loads,ans,kdiag)
    ans(neq)=ans(neq)+f1; ans(neq-1)=ans(neq-1)+f2
    CALL spabac(bp,ans,kdiag); loads=ans
    IF(nod==4.AND.j==ntime)CALL contour(loads,g_coord,g_num,nci,argv,nlen,13)
    IF(j/npri*npri==j)WRITE(11,'(2e12.4)')
        & time,loads(nres)*exp(-ux*g_coord(1,nres)/two/kay(1,1))*&
        & exp(-uy*g_coord(2,nres)/two/kay(2,2))
END DO timesteps
STOP
END PROGRAM p89

```

When convection terms are retained in the simplified flow equations, (2.139) has to be solved. Again many techniques could be employed but, in the present program, an implicit algorithm based on equation (3.100) is used. Thus this program is an extension of Program 8.4.

When the transformation of equation (2.141) is employed, the equation to be solved becomes

$$c_x \frac{\partial^2 h}{\partial x^2} + c_y \frac{\partial^2 h}{\partial y^2} - \left( \frac{u^2}{4 c_x} + \frac{v^2}{4 c_y} \right) h = \frac{\partial h}{\partial t} \quad (8.3)$$

thus the extra term involving  $h$  distinguishes the process from a simple diffusion one. However, reference to Table 2.1 shows that the semi-discretised ‘stiffness’ matrix for this problem will still be symmetrical, the  $h$  term involving an element matrix of the ‘mass matrix’ type, namely  $\int \int N_i N_j dx dy$ .

Comparison with Program 8.4 will show essentially the same array declarations and input parameters. Extra variables are the velocities  $u$  and  $v$  in the  $x$ - and  $y$ -directions,  $ux$  and  $uy$ , respectively.

The problem chosen is the one-dimensional example shown in Figure 8.26, consisting of a 56 m deep bed of fluid discretised by 40, 4-node elements. There is a steady velocity in the  $y$ -direction ( $uy$ ) of 0.0135, and the initial concentration at all points is set to zero. The dependent variable  $\phi$  refers to the concentration of sediment picked up by the flow from the base of the mesh ( $y = 0.0$ ), and distributed with time in the  $y$ -direction. The velocity in the  $x$ -direction ( $ux$ ) is zero, and for numerical reasons  $c_x$  is set to a small number,  $1 \times 10^{-6}$ , which is effectively zero. After reading the mesh coordinate data  $x\_coords$  and  $y\_coords$ , the implicit time-stepping parameters  $dtim$ ,  $nsteps$  and  $theta$  are read, followed by the output control parameters  $npri$ ,  $nres$  and  $ntime$ . In this example, a fully ‘implicit’ time-stepping scheme is illustrated by setting  $theta=1$ . The output calls for results to be printed every time step at node 82. The steady velocities are read as  $ux$  and  $uy$  and the contour map of concentration after  $ntime$  time steps will be written to file  $*.con$  with  $nci$  contour intervals.

The equation to be solved reduces to,

$$c_y \frac{\partial^2 h}{\partial y^2} - \frac{v^2}{4 c_y} h = \frac{\partial h}{\partial t} \quad (8.4)$$

subject to the boundary conditions at  $y = 0$  of

$$\frac{\partial \phi}{\partial y} = \frac{v}{c_y} = C_2 \quad (8.5)$$

and at  $y = 56.0$  of

$$\frac{\partial \phi}{\partial y} = \frac{v}{c_y} \phi = C_2 \phi \quad (8.6)$$

which after transformation, become

$$\frac{\partial h}{\partial y} = -\frac{v}{2c_y} h + \frac{v}{c_y} \quad (8.7)$$

and

$$\frac{\partial h}{\partial y} = \frac{v}{2c_y} h \quad (8.8)$$

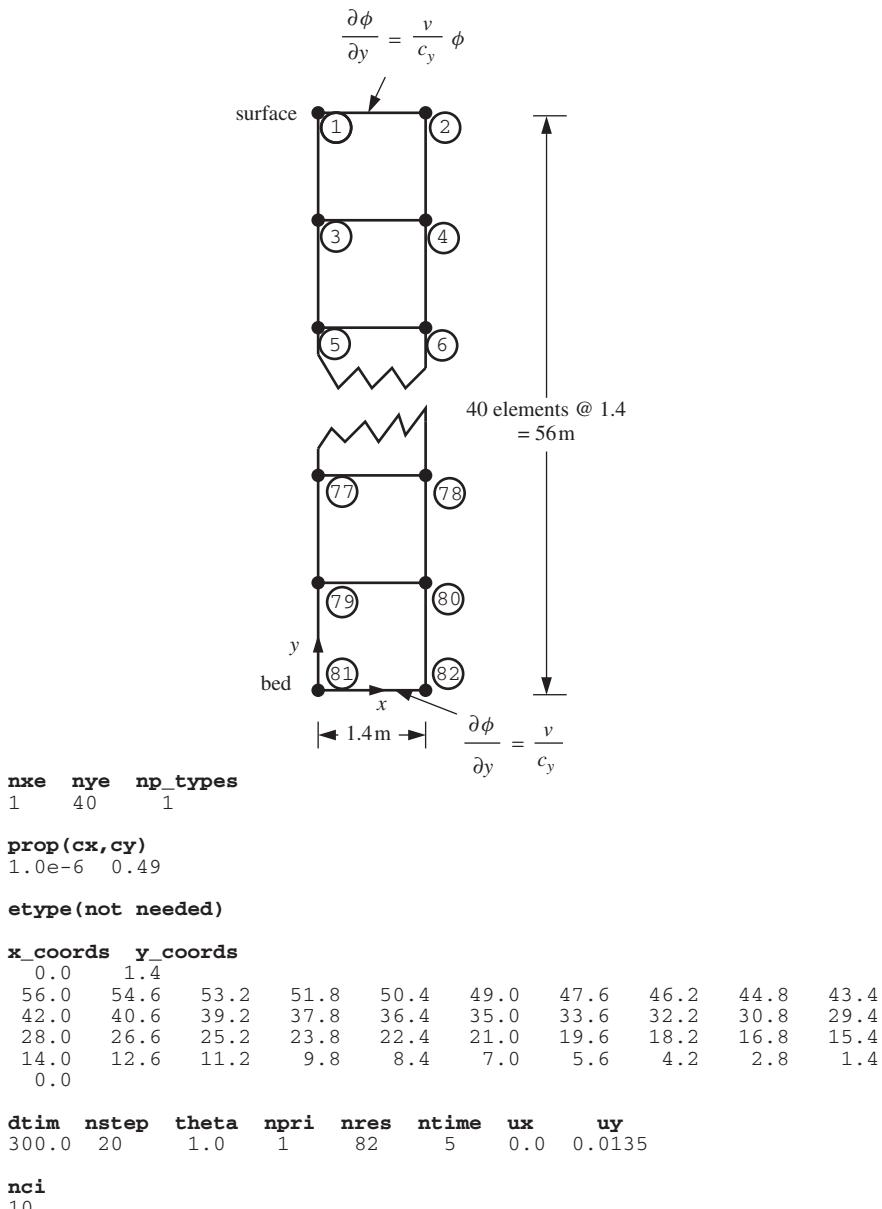


Figure 8.26 Mesh and data for Program 8.9 example

Boundary condition (8.8) is clearly of the type described in Section 3.6, equation (3.33), hence at that boundary, the element matrix will have to be augmented by the matrix shown in equation (3.37). The multiple  $C_1 c_y (x_k - x_j)/6$  in (3.37) is just  $v(x_k - x_j)/12$  or  $uy * (x_coords(2) - x_coords(1)) / d12$  in the program. This is carried out in the section of program headed ‘derivative boundary conditions’.

There are 82 equations and the skyline storage is 283

Time	Concentration (node 82)
0.0000E+00	0.0000E+00
0.3000E+03	0.2828E+00
0.6000E+03	0.4011E+00
0.9000E+03	0.4796E+00
0.1200E+04	0.5389E+00
0.1500E+04	0.5867E+00
0.1800E+04	0.6269E+00
0.2100E+04	0.6616E+00
0.2400E+04	0.6920E+00
0.2700E+04	0.7192E+00
0.3000E+04	0.7435E+00
0.3300E+04	0.7656E+00
0.3600E+04	0.7856E+00
0.3900E+04	0.8038E+00
0.4200E+04	0.8204E+00
0.4500E+04	0.8356E+00
0.4800E+04	0.8494E+00
0.5100E+04	0.8621E+00
0.5400E+04	0.8737E+00
0.5700E+04	0.8843E+00
0.6000E+04	0.8941E+00

**Figure 8.27** Results from Program 8.9 example

The condition (8.7) contains a similar contribution, but in addition the term  $v/c_y$  is of the type described by equation (3.34). Thus, an addition must be made to the right-hand side of the equations at such a boundary in accordance with equation (3.39). In this case the terms in equation (3.39) are  $v(x_k - x_j)/2$  and are incorporated in the program immediately after the comment ‘factorise equations’.

This example shows that quite complicated coding would be necessary to permit very general boundary conditions to be specified in all problems.

After insertion of boundary conditions the (constant) global left-hand side matrix (*bp*) is factorised using *sparin*. The time-stepping loop follows a familiar course, with matrix-by-vector multiplication using subroutine *linmul\_sky* followed by forward and back substitution using *spabac*. Output for 20 steps is listed in Figure 8.27, while Figure 8.28 shows how the finite element solution compares with an ‘analytical’ one due to Dobbins (1944).

It should be remembered that solutions are in terms of the transformed variable *h*, and the true solution  $\phi$  has been recovered using (2.141).

### Program 8.10 Plane analysis of the diffusion-convection equation using 4-node rectangular quadrilaterals. Implicit time integration using the ‘theta’ method. Untransformed solution

```

PROGRAM p810
!-----
! Program 8.10 Plane analysis of the diffusion-convection equation
!           using 4-node rectangular quadrilaterals. Implicit time
!           integration using the "theta" method.
!           Untransformed solution.
!-----
USE main;  USE geom; IMPLICIT NONE

```

```

INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,j,k,nband,ndim=2,nels,neq,fixed_freedoms,nip=4,nlen,nn,    &
nod=4,npri,np_types,nres,nstep,ntime,nxe,nye
REAL(iwp)::det,dtim,part1,part2,pt2=0.2_iwp,penalty=1.0e20_iwp,theta,      &
time,ux,uy,zero=0.0_iwp; CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,),g_num(:,:,),node(:,),num(:)
REAL(iwp),ALLOCATABLE::ans(:,),conc(:,),coord(:, :,),copy(:, :,),der(:, :,),
deriv(:, :,),dtkd(:, :,),fun(:,),g_coord(:, :,),jac(:, :,),kb(:, :,),kc(:, :,),
loads(:,),ntn(:, :,),pb(:, :,),mm(:, :,),points(:, :,),prop(:, :,),storpby(:, ),
weights(:,),work(:, :,),x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
ALLOCATE(points(nip,ndim),weights(nip),coord(nod,ndim),fun(nod),          &
jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod),          &
mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod),          &
dtkd(nod,nod),prop(ndim,np_types),x_coords(nxe+1),y_coords(nye+1),          &
etype(nels),conc(nye+1))
READ(10,*)prop; etype=1; if(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords
READ(10,*)dtim,nstep,theta,npri,nres,ntime,ux,uy; nband=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
    IF(nband<bandwidth(num))nband=bandwidth(num)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
ALLOCATE(kb(neq,2*nband+1),pb(neq,2*nband+1),work(nband+1,neq),          &
copy(nband+1,neq),loads(0:neq),ans(0:neq))
WRITE(11,'(2(a,i5))')                                     &
" There are",neq," equations and the half-bandwidth is",nband
CALL sample(element,points,weights); kb=zero; pb=zero
!-----global conductivity and "mass" matrix assembly-----
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der)
        DO j=1,nod; DO k=1,nod
            part1=prop(1,etype(iel))*deriv(1,j)*deriv(1,k)+          &
            prop(2,etype(iel))*deriv(2,j)*deriv(2,k)
            part2=ux*fun(j)*deriv(1,k)+uy*fun(j)*deriv(2,k)
            dtkd(j,k)=(part1-part2)*det*weights(i)
        END DO; END DO
        kc=kc+dtkd; CALL cross_product(fun,fun,ntn); mm=mm+ntn*det*weights(i)
    END DO gauss_pts; mm=mm/(theta*dtim)
    CALL formtb(kb,kc,num); CALL formtb(pb,mm,num)
END DO elements_2; pb=pb+kb; kb=pb-kb/theta
!-----boundary conditions-----
READ(10,*)fixed_freedoms
ALLOCATE(node(fixed_freedoms),storpby(fixed_freedoms))
READ(10,*)node; pb(node,nband+1)=pb(node,nband+1)+penalty
storpby=pb(node,nband+1)
!-----factorise equations-----

```

```

work=zero; CALL gauss_band(pb,work)
WRITE(11,'(/a,i3,a)')      Time      Concentration(node",nres,")
WRITE(11,'(2e12.4)')0.0,loads(nres); loads=zero
!-----time stepping loop-----
timesteps: DO j=1,nstep
    time=j*dtim; copy=work; CALL bantmul(kb,loads,ans); ans(0)=zero
    IF(time<=pt2)THEN; ans(node)=storp; ELSE; ans(node)=zero; END IF
    CALL solve_band(pb,copy,ans); ans(0)=zero; loads=ans
    IF(j==ntime)conc(:)=loads(2:neq:2)
    IF(j/npri*npri==j)WRITE(11,'(2e12.4)')time,loads(nres)
END DO timesteps
WRITE(11,'(/a,e10.4,a)')
    " Distance      Concentration(time=",nstep*dtim,")"
DO i=1,nye+1; WRITE(11,'(2e12.4)')y_coords(i),conc(i); END DO
STOP
END PROGRAM p810

```

In the previous example, difficulties would have arisen had the ratio of  $u$  to  $c_x$  been large, because the transformation involving  $\exp(u/c_x)$  would not have been numerically

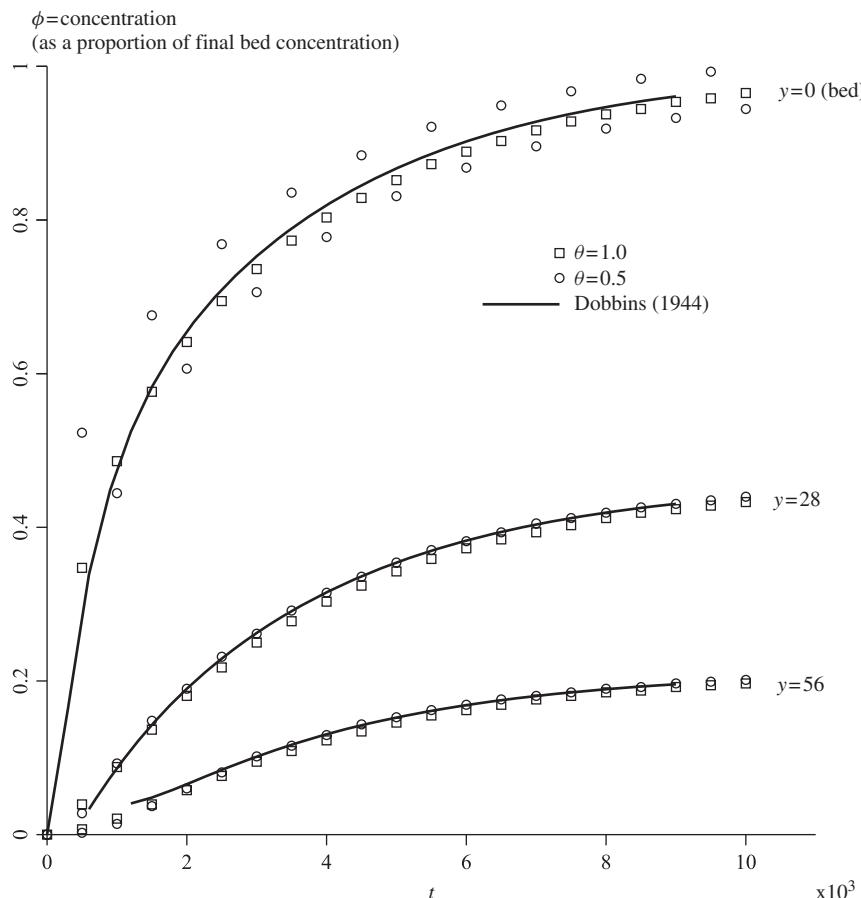
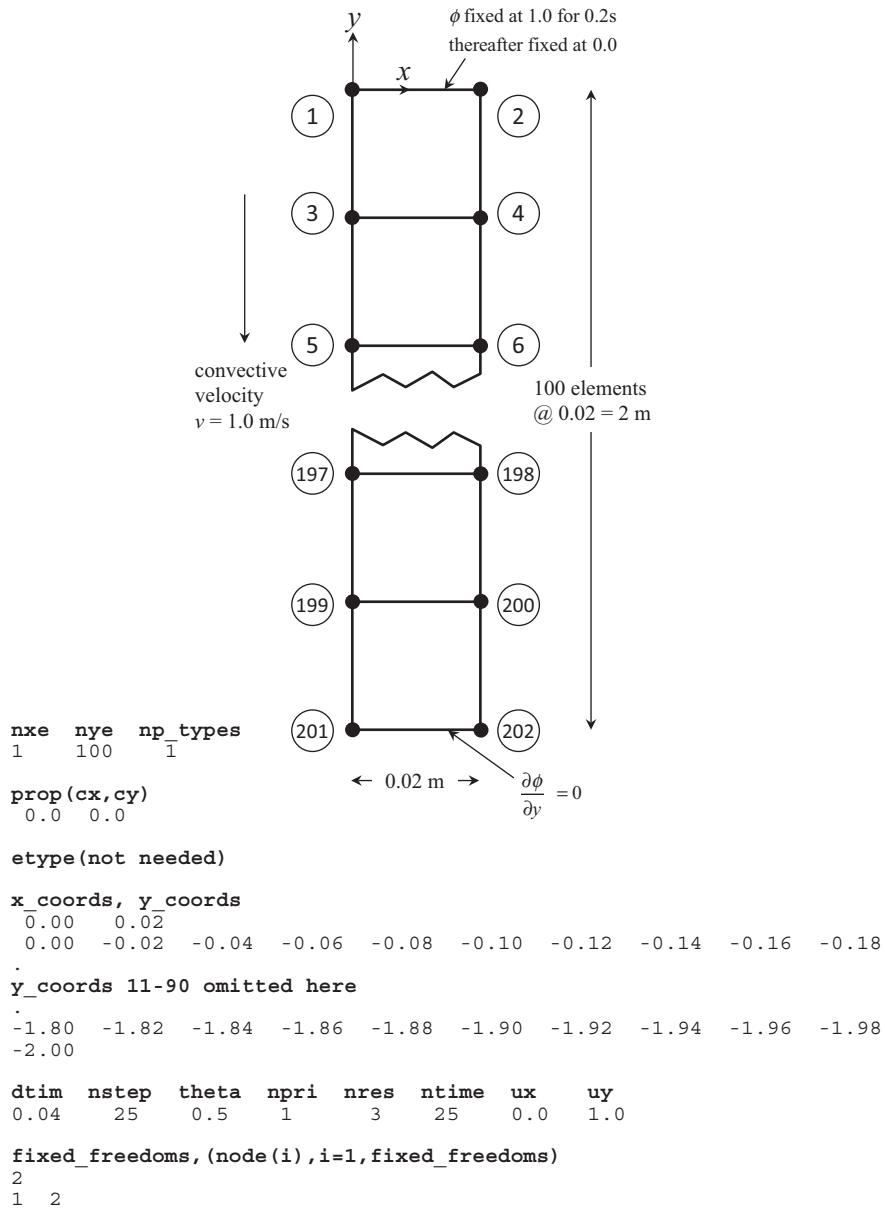


Figure 8.28 Graph of concentration vs. time from Program 8.9 example

feasible (Smith *et al.*, 1973). Under these circumstances (and even when  $c_x = c_y = 0$ ), equation (2.139) can still be solved, but with the drawback that the element and system matrices become unsymmetrical, although the latter are still banded.

Program 8.10 will be used to solve a purely convective problem, that is with  $c_x = c_y = 0$ . The problem chosen is again one-dimensional, as shown in Figure 8.29,



**Figure 8.29** Mesh and data for Program 8.10 example

so the equation is effectively

$$-v \frac{\partial \phi}{\partial y} = \frac{\partial \phi}{\partial t} \quad (8.9)$$

in the region  $0 \leq y \leq -2$ , subject to the boundary conditions  $\phi = 1$  at  $y = 0$  for  $0 \leq t \leq 0.2$ ,  $\phi = 0$  at  $y = 0$  for  $t > 0.2$  and  $\partial\phi/\partial y = 0$  at  $y = -2$  for all  $t$ .

Comparing with Program 8.9, the usual geometry subroutine for 4-node elements numbered in the  $x$ -direction, `geom_rect` with '`dir = 'x'`' is used. Arrays `node` and `storp` are used to read in the numbers of fixed freedoms and to store information

```
There are 202 equations and the half-bandwidth is      3

Time      Concentration(node  3)
0.0000E+00  0.0000E+00
0.4000E-01  0.3660E+00
0.8000E-01  0.1116E+01
0.1200E+00  0.1066E+01
0.1600E+00  0.9285E+00
0.2000E+00  0.1017E+01
(some results omitted here)
0.8800E+00  -0.6148E-02
0.9200E+00  -0.3116E-02
0.9600E+00  0.8138E-02
0.1000E+01  -0.4934E-02

Distance    Concentration(time=0.1000E+01)
0.0000E+00  0.3289E-24
-0.2000E-01  -0.4934E-02
-0.4000E-01  0.1548E-01
-0.6000E-01  -0.4225E-02
-0.8000E-01  -0.2975E-01
-0.1000E+00  0.3345E-01
(some results omitted here)
-0.7000E+00  -0.9004E-01
-0.7200E+00  0.1104E+00
-0.7400E+00  0.3613E+00
-0.7600E+00  0.6202E+00
-0.7800E+00  0.8459E+00
-0.8000E+00  0.1009E+01
-0.8200E+00  0.1095E+01
-0.8400E+00  0.1105E+01
-0.8600E+00  0.1051E+01
-0.8800E+00  0.9513E+00
-0.9000E+00  0.8246E+00
-0.9200E+00  0.6881E+00
-0.9400E+00  0.5551E+00
-0.9600E+00  0.4344E+00
-0.9800E+00  0.3306E+00
-0.1000E+01  0.2453E+00
-0.1020E+01  0.1779E+00
-0.1040E+01  0.1262E+00
-0.1060E+01  0.8776E-01
-0.1080E+01  0.5989E-01
-0.1100E+01  0.4017E-01
-0.1120E+01  0.2650E-01
(some results omitted here).
-0.1900E+01  0.1688E-11
-0.1920E+01  0.8274E-12
-0.1940E+01  0.4025E-12
-0.1960E+01  0.1973E-12
-0.1980E+01  0.9360E-13
-0.2000E+01  0.4779E-13
```

**Figure 8.30** Results from Program 8.10 example

about them during the time-stepping process. The system  $\mathbf{kb}$  matrix is now unsymmetrical and stored as a rectangular array with the full bandwidth using subroutine `formtb` (see Figure 3.20). Although  $\mathbf{pb}$  is symmetrical, it too is stored as a full band to be compatible with  $\mathbf{kb}$ .

In the element integration and assembly loop, `part1` accumulates the diffusive part of the element ‘stiffness’ and `part2` the convective part. The `part2` contribution to the  $\mathbf{kc}$  matrix is the only one in the present example and it is unsymmetrical (skew-symmetrical in fact).

The structure of the program is modelled on the previous one. The solution routines are `gauss_band` and `solve_band` which use an extra array `work` as working space. Matrix-by-vector multiplication needs the subroutine `bantmul` (see Tables 3.4 and 3.5).

In the section ‘time-stepping loop’ it can be seen that the solution at nodes 1 and 2 is held at the value 1.0 for the first 0.2 s of convection and at zero subsequently.

After reading the mesh coordinate data `x_coords` and `y_coords`, the conventional implicit time-stepping parameters `dtime`, `nsteps` and `theta` are read, followed by the output control parameters. The output calls for results to be printed every time step at node 3. No contour map is produced in this case, but instead the output produces the spatial concentration after `ntime=25` time steps. The velocities are read as `ux=0.0` and `uy=1.0`, followed by the number of fixed freedoms `fixed Freedoms`, and the node numbers to be fixed.

The variation of concentration with time at node 3 and the concentration distribution after 1.0 s are shown in the results file in Figure 8.30. Figure 8.31 gives a plot of the computed solution after 1 s together with the correct solution to the problem as described by a rectangular pulse moving with unit velocity in the  $y$ -direction. Spurious spatial oscillations are seen to have been introduced by the numerical solution. Measures to improve the solutions are beyond the scope of the present treatment (see Smith, 1976, 1979). Of course in the present case, improvements can be achieved by simply reducing the element size in the  $y$ -direction and the time step size  $\Delta t$ .

## **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**

```
PROGRAM p811
!-----
! Program 8.11 Plane or axisymmetric thermal conduction analysis using
!           4-node rectangular quadrilaterals. Option of convection and
!           flux boundary conditions.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,j,k,nci,ncon,ndim=2,nels,neq,nflx,nip=4,      &
nlen,nn,nod=4,nodl=2,npri,nprops=3,np_types,nres,nstep,ntime,nxe,nye
REAL(iwp)::det,dtime,d3=3.0_iwp,d4=4.0_iwp,d6=6.0_iwp,d12=12.0_iwp,      &
hpl,one=1.0_iwp,length,penalty=1.0e20_iwp,pt5=0.5_iwp,theta,time,      &
two=2.0_iwp,zero=0.0_iwp
CHARACTER(len=15)::argv,dir,element='quadrilateral',type_2d
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::econn(:,:),etype(:),g_num(:,:,1),node(:,1),num(:,1),kdiag(:)
```

```

REAL(iwp,ALLOCATABLE::bp(:),coord(:,:,der(:,:,deriv(:,:,econv(:,:, &
    flxv(:),fun(:),gc(:),g_coord(:,:,jac(:,:,kay(:,:,kc(:,:,kcl(:,:, &
    kv(:),loads(:),newlo(:),ntn(:,:,mm(:,:,points(:,:,prop(:,:,rhs(:), &
    storbp(:),value(:),weights(:),x_coords(:),y_coords(:) &

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)type_2d,dir,nxe,nye,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); neq=nn
ALLOCATE(points(nip,ndim),weights(nip),kay(ndim,ndim),coord(nod,ndim), &
    fun(nod),jac(ndim,ndim),g_coord(ndim,nn),der(ndim,nod),deriv(ndim,nod), &
    mm(nod,nod),g_num(nod,nels),kc(nod,nod),ntn(nod,nod),num(nod), &
    etype(nels),kdiag(neq),loads(0:neq),newlo(0:neq),flxv(0:neq), &
    x_coords(nxe+1),y_coords(nye+1),prop(nprops,np_types),gc(ndim), &
    kcl(nodl,nodl),rhs(0:neq))
READ(10,*)prop; etype=1; if(np_types>1)read(10,*)etype
READ(10,*)x_coords,y_coords
READ(10,*)dtim,nstep,theta,npri,nres,ntime; kdiag=0
! -----loop the elements to set up global geometry and kdiag -----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,dir)
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
    CALL fkdiag(kdiag,num)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),bp(kdiag(neq)))
WRITE(11,'(2(A,I5))') &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); bp=zero; kv=zero; gc=one
!-----global conductivity and "mass" matrix assembly-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); kc=zero; mm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); if(type_2d=='axisymmetric')gc=MATMUL(fun,coord)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)*gc(1)
        CALL cross_product(fun,fun,ntn)
        mm=mm+ntn*det*weights(i)*gc(1)*prop(3,etype(iel))
    END DO gauss_pts
    CALL fsparv(kv,kc,num,kdiag); CALL fsparv(bp,mm,num,kdiag)
END DO elements_2
flxv=zero; READ(10,*)nflx,(k,flxv(k),i=1,nflx);rhs=flxv
READ(10,*)ncon; IF(ncon/=0)then; ALLOCATE(econn(nodl,ncon),econv(2,ncon))
    READ(10,*)(econn(:,i),econv(:,i),i=1,ncon)
    DO i=1,ncon
        length=SQRT((g_coord(1,econn(1,i))-g_coord(1,econn(2,i)))**2 &
            +(g_coord(2,econn(1,i))-g_coord(2,econn(2,i)))**2)
        hpl=econv(1,i)*length
        IF(type_2d=='axisymmetric')THEN
            rhs(econn(1,i))=rhs(econn(1,i))+hpl*econv(2,i)* &
                (two*g_coord(1,econn(1,i))+g_coord(1,econn(2,i)))/d6
            rhs(econn(2,i))=rhs(econn(2,i))+hpl*econv(2,i)* &
                (two*g_coord(1,econn(2,i))+g_coord(1,econn(1,i)))/d6
            kcl(1,1)=hpl*(g_coord(1,econn(1,i))/d4+g_coord(1,econn(2,i))/d12)
            kcl(1,2)=hpl*(g_coord(1,econn(1,i))+g_coord(1,econn(2,i)))/d12
            kcl(2,2)=hpl*(g_coord(1,econn(2,i))/d4+g_coord(1,econn(1,i))/d12)
        END IF
    END DO
END IF

```

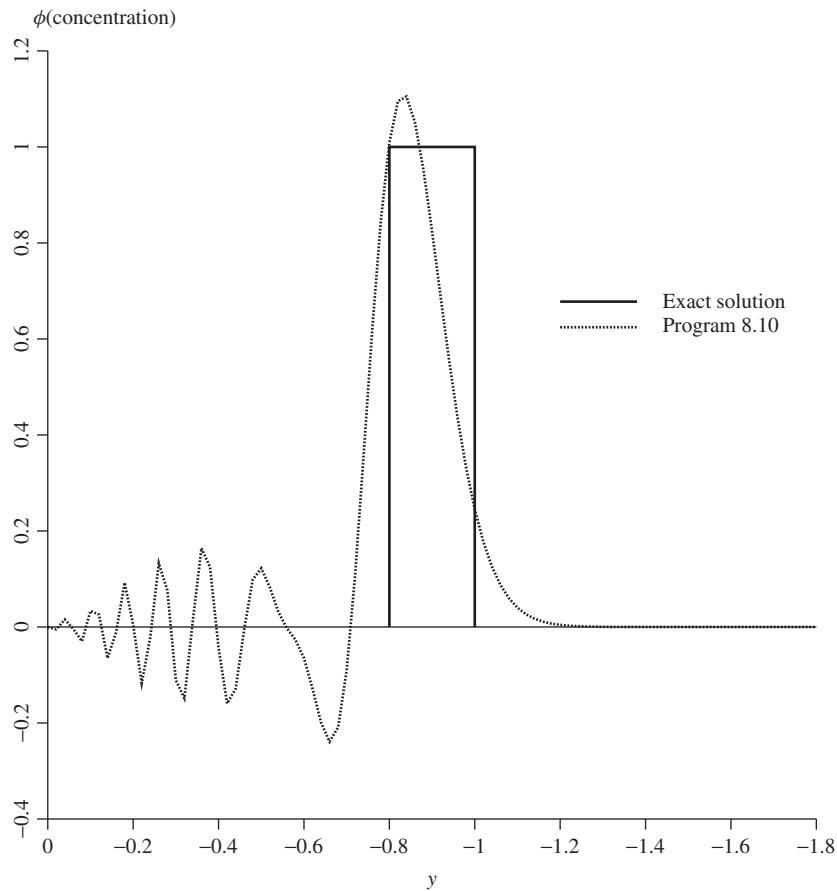
```

kcl(2,1)=kcl(1,2)
ELSE
rhs(econn(:,i))=rhs(econn(:,i))+hpl*econv(2,i)*pt5
kcl(1,1)=hpl/d3; kcl(1,2)=hpl/d6
kcl(2,1)=kcl(1,2); kcl(2,2)=kcl(1,1)
ENDIF; CALL fsparv(kv,kcl,econn(:,i),kdiag)
ENDDO
END IF
kv=kv*theta*dtim; bp=bp+kv; kv=bp-kv/theta; rhs=rhs*dtim
!-----specify initial and boundary values-----
READ(10,*) loads(1:); loads(0)=zero; READ(10,*) fixed_freedoms
IF(fixed_freedoms/=0)then
    ALLOCATE(node(fixed_freedoms),value(fixed_freedoms),
              storbp(fixed_freedoms))
    READ(10,*)(node(i),value(i),i=1,fixed_freedoms)
    bp(kdiag(node))=bp(kdiag(node))+penalty; storbp=bp(kdiag(node))
END IF
!-----factorise equations-----
CALL sparin(bp,kdiag)
!-----time stepping loop-----
WRITE(11,'(/A,I3,A)')" Time Pressure (node",nres,")"
WRITE(11,'(2E12.4)')0.0,loads(nres)
timesteps: DO j=1,nstep
    time=j*dtim; CALL linmul_sky(kv,loads,newlo,kdiag); newlo=newlo+rhs
    IF(fixed_freedoms/=0)newlo(node)=storbp*value
    CALL spabac(bp,newlo,kdiag); loads=newlo
    IF(node==4.AND.j==ntime)THEN; READ(10,*)nci
        CALL contour(loads,g_coord,g_num,nci,argv,nlen,13)
    END IF
    IF(j/npri*npri==j)WRITE(11,'(2E12.4)')time,loads(nres)
END DO timesteps
STOP
END PROGRAM p811

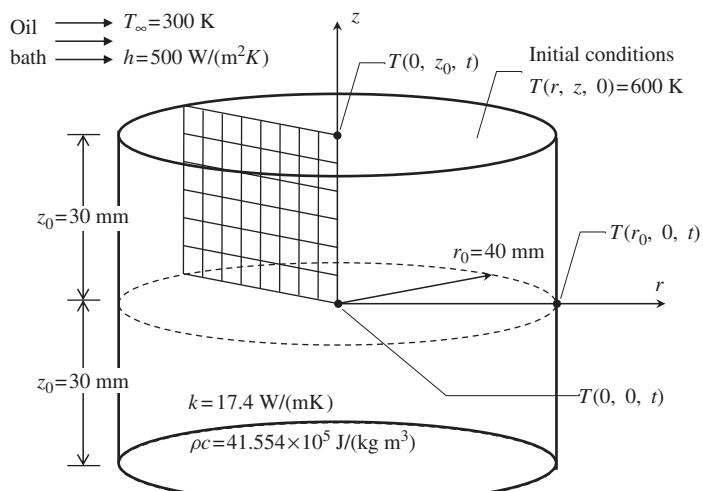
```

The program is an extension of Program 8.4 with the additional options of steady flux input and convection boundary conditions. A distributed flux on the boundary is input as equivalent nodal flux values, in the same way as an external pressure in the elastic analyses of Chapter 5 would be read in as equivalent nodal loads using the relationships given in Appendix A. In this program *nflx* is the number of nodes to be subjected to flux ‘loading’ and *flxv* is a global vector holding the values of the nodal fluxes.

The convection boundary conditions of the type discussed in Section 3.6.1 start by reading *ncon*, which represents the number of element sides that will receive convection boundary conditions. For each of the *ncon* sides, two INTEGER followed by two REAL numbers are read. The INTEGERs representing the (global) node numbers of each element side are read into *econn*, and the REALs representing the convection heat transfer coefficient *h* and the ambient outside temperature  $T_\infty$  are read into *econv*. The convection boundary conditions involve modifications to both the left- and right-hand sides of the conduction equations as discussed in (3.40)–(3.45). The left-hand-side modifications are held locally in array *kcl* before assembly into the global left-hand-side matrix *kv*, and the right-hand-side modifications are added directly into the global right-hand-side vector *rhs*. The modifications account for plane or axisymmetric conditions as needed. Other variables used by this program include *hpl* and *length*, where *hpl* holds the product of the convection heat transfer coefficient and the element side length *length*.



**Figure 8.31** Concentration vs. distance after 1 s from Program 8.10 example



**Figure 8.32** Example problem of a hot steel cylinder quenched in oil

```

type_2d
'axisymmetric'

nod      dir
8      'z'

nre      nze      np_types
8      6      1

prop(kx,ky,ρc)
17.4  17.4  4155400.0

etpye (not needed)

x_coords  y_coords
0.00  0.005  0.01  0.015  0.02  0.025  0.03  0.035  0.04
0.0 -0.005 -0.01 -0.015 -0.02 -0.025 -0.03

dtime    nstep    theta    npri    nres    ntime
1.0      180      0.5      15       7       180

nflx, (k, flxv(k), i=1, nflx)
0

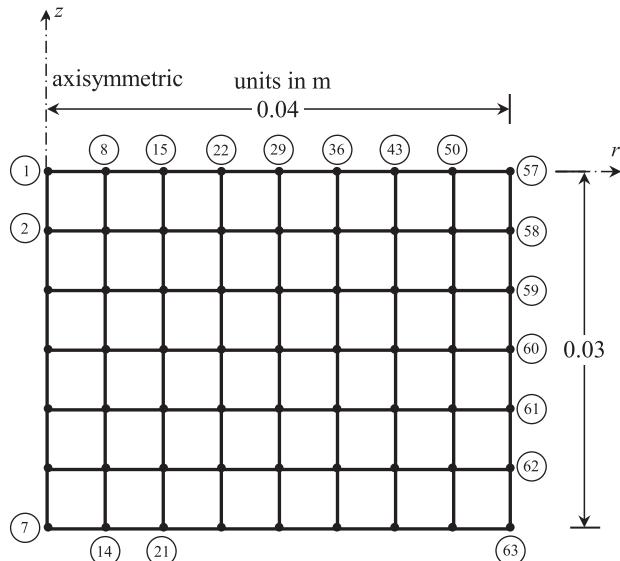
ncon, (econn(2,i), econv(2,i), i=1, ncon)
14
  1   8   500.0  300.0      8   15   500.0  300.0
  15  22  500.0  300.0     22  29   500.0  300.0
  29  36  500.0  300.0     36  43   500.0  300.0
  43  50  500.0  300.0     50  57   500.0  300.0
  57  58  500.0  300.0     58  59   500.0  300.0
  59  60  500.0  300.0     60  61   500.0  300.0
  61  62  500.0  300.0     62  63   500.0  300.0

loads(i), i=1, neq
600.0  600.0 . . loads(3:61) omitted here . . 600.0  600.0

fixed_freedoms, (node(i), value(i), i=1, fixed_freedoms)
0

nci
10

```



**Figure 8.33** Mesh and data for Program 8.11 example

The example shown in Figure 8.32 is described by Incropera and DeWitt (2002) and involves a stainless steel cylinder, initially at 600 K, quenched by submersion in an oil bath maintained at 300 K with convection heat transfer coefficient  $h = 500 \text{ W/m}^2$ . The cylinder has a height of 0.06 m and a diameter of 0.08 m. Other properties involve an (isotropic) thermal conductivity  $k = 17.4 \text{ W/(mK)}$ , mass density  $\rho = 7900 \text{ kg/m}^3$  and specific heat  $c = 526 \text{ J/(kg K)}$ . The product of the mass density and specific heat  $\rho c = 41.554 \times 10^5 \text{ J/(kg m}^3)$  is typically combined as a single property. The thermal diffusivity of the steel, given by  $\alpha = k/\rho c = 4.19 \times 10^{-6} \text{ m}^2/\text{s}$  is computed internally by the program. The example is axisymmetric, and has an additional horizontal plane of symmetry at mid-height, so an axisymmetric analysis of the top half is performed using the mesh and data indicated in Figure 8.33.

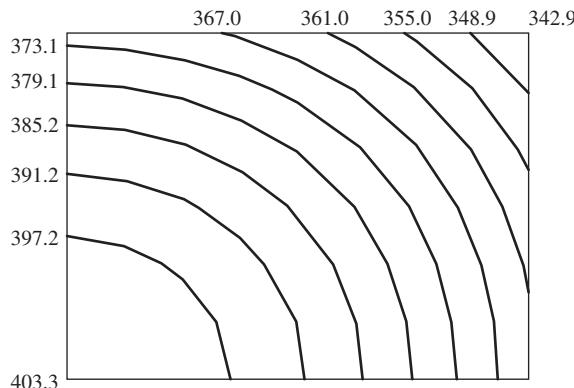
The data follows a familiar course with rectangular mesh generation using 4-node quadrilaterals and three properties assigned to each element, namely the thermal conductivities in the  $x$ - and  $y$ -directions ( $k_x, k_y$ ) and the product property ( $\rho c$ ) as mentioned above. An implicit time-stepping procedure with 180 steps of  $\Delta t = 1 \text{ s}$  is implemented, with the temperature at node 7 (the centre of the cylinder) output every 15 s. This example has no external flux applied, so `nflux` is read as zero and no further nodal flux data is required. The top and right sides of the mesh are subjected to convection boundary conditions involving 14 element sides, so `ncon` is read as 14 followed, for each side, by the global node number pairs, the convection heat transfer coefficient ( $h$ ) and the ambient external temperature ( $T_\infty$ ). All nodes are initially set to a temperature of 600 K. The program allows the user to specify fixed temperature boundary conditions, but none is required in this example, so `fixed_freedom` is set to zero. A contour map with 10 contour intervals (`nci=10`) is requested after 180 time steps (`ntime=180`).

The results file shown in Figure 8.34 indicates that the temperature at the centre of the cylinder falls to 403.3 K after 3 min, which can be compared with 405 K obtained by Incropera and DeWitt (2002) using a semi-analytical approach involving products of 1D solutions. Figure 8.35 shows a contour map of temperature after 3 min. It may be noted that the top right corner of the mesh has a temperature of 342.9 K, which again can be compared with the product solution of 344 K. As expected, the temperature contours are normal

There are 63 equations and the skyline storage is 509

Time	Pressure (node 7)
0.0000E+00	0.6000E+03
0.1500E+02	0.5999E+03
0.3000E+02	0.5928E+03
0.4500E+02	0.5749E+03
0.6000E+02	0.5518E+03
0.7500E+02	0.5277E+03
0.9000E+02	0.5044E+03
0.1050E+03	0.4829E+03
0.1200E+03	0.4634E+03
0.1350E+03	0.4457E+03
0.1500E+03	0.4300E+03
0.1650E+03	0.4159E+03
0.1800E+03	0.4033E+03

**Figure 8.34** Results from Program 8.11 example



**Figure 8.35** Contours of temperature (in K) after 3 min from Program 8.11 example

to the axes of symmetry on the left and bottom surfaces, while the convection boundary conditions lead to inclined intersections with the contours on the top and right surfaces.

It was noted that if the convection heat transfer coefficient ( $h$ ) or the ambient temperature ( $T_\infty$ ) is to change at any point on the boundary (i.e., two neighbouring elements have different convection boundary conditions) then better numerical results were obtained if the off-diagonal terms in the left-hand-side matrices corresponding to those elements given in (3.42) and (3.44) were set to zero.

### 8.3 Glossary of Variable Names

#### Scalar integers

cg_iters	pcg iteration counter
cg_limit	pcg iteration ceiling
fixed_freedoms	number of fixed total heads
i, iel	simple counters
iwp	SELECTED_REAL_KIND(15)
j, k	simple counters
nband	full bandwidth of non-symmetric matrix
nci	number of contour intervals
ncon	number of element sides with convection boundary conditions
ndim	number of dimensions
nels	number of elements
neq	number of degrees of freedom in the mesh
nflux	number of nodes with input flux values
nip	number of integrating points
nlen	maximum number of characters in data file basename
nn	number of nodes
nod	number of nodes per element
nodl	number of nodes on element side (set to 2)
npri	output printed every npri time steps
nprops	number of material properties

np_types	number of different property types
nres	node number at which time history is to be printed
nstep	number of time steps required
ntime	time step number at which spatial distribution is to be printed or contoured
nxe,nye	number of columns and rows of elements
<b>Scalar reals</b>	
alpha	$\alpha$ from equations (3.22)
at	holds area beneath isochrone by trapezoid rule at time $t$
a0	holds area beneath isochrone by trapezoid rule at time $t = 0$
beta	$\beta$ from equations (3.22)
cg_tol	pcg convergence tolerance
det	determinant of the Jacobian matrix
dtim	calculation time step
d3,d4,d6,d12	set to 3.0, 4.0, 6.0 and 12.0
f1,f2	used to fix derivative boundary conditions
gamw	unit weight of water
hpl	product of heat transfer coefficient and side length
length	element side length
one	set to 1.0
part1,part2	diffusive and convective parts
penalty	set to $1 \times 10^{20}$
pt2,pt25,pt5	set to 0.2, 0.25 and 0.5
sc	ultimate consolidation settlement
s1	consolidation settlement still left to occur
theta	time-integration weighting parameter
time	holds elapsed time $t$
two	set to 2.0
uav	average degree of consolidation based on excess pore pressure dissipation
uavs	average degree of consolidation based on settlement
up	holds dot product $\{\mathbf{R}\}_k^T \{\mathbf{R}\}_k$ from equations (3.22)
ux,uy	velocities in $x$ - and $y$ -directions
zero	set to 0.0
<b>Scalar characters</b>	
argv	holds data file basename
dir	element and node numbering direction
element	element type
type_2d	type of 2D analysis
<b>Scalar logical</b>	
cg_converged	set to .TRUE. if pcg process has converged
<b>Dynamic integer arrays</b>	
econn	side node numbers for convection boundaries conditions
etype	element property types
g_num	node numbers for all elements

kdiag	diagonal term locations
node	nodes with fixed total heads
num	element node numbers
<b>Dynamic real arrays</b>	
ans	rhs vector in equilibrium equations
bp	global ‘mass’ matrix
conc	concentration distribution after ntime steps
coord	element nodal coordinates
copy	working space array
d	vector used in coding of equations (3.22)
der	shape function derivatives with respect to local coordinates
deriv	shape function derivatives with respect to global coordinates
diag_precon	diagonal preconditioner vector
econv	values of heat transfer coefficients $h$ and ambient temperatures $T_\infty$
ell	element lengths
flxv	global vector of boundary nodal flux values
fun	shape functions
gc	integrating point coordinates
globma	global lumped mass matrix (stored as a vector)
g_coord	nodal coordinates for all elements
jac	Jacobian matrix
kay	permeability matrix
kc	element conductivity matrix
kcl	left-hand-side matrices for convection boundary conditions
kv	global conductivity matrix
loads	excess pore pressure values
mass	element lumped mass vector
mm	element ‘mass’ matrix
newlo	new excess pore pressure values
ntn	cross product of shape functions
p	‘descent’ vector used in equations (3.22)
points	integrating point local coordinates
press	excess pore pressure values after ntime time steps
prop	element properties
r	holds fixed rhs terms in pcg solver
rhs	right-hand-side vectors for convection and flux boundary
storpb	stores augmented diagonal terms
store	stores global augmented diagonal terms
storka, storkb	stores lhs and rhs element matrices
store_kc	stores element kc matrices
store_mm	stores lhs element matrices
u	vector used in equations (3.22)
value	fixed boundary values of excess pore pressure

weights	weighting coefficients
work	working space array
x, xnew	'old' and 'new' solution vector
x_coords, y_coords	$x(r)$ - and $y(z)$ -coordinates of mesh layout

## 8.4 Exercises

1. A layer of clay of thickness  $2D$ , free draining at its top and bottom surfaces is subjected to a suddenly applied distributed load of one unit. Working in terms of a dimensionless time factor given by  $T = c_v t / D^2$ , and using a single finite element, use the Crank–Nicolson approach ( $\theta = 0.5$ ) with a time step of  $\Delta T = 0.1$  to estimate the mid-depth pore pressure when  $T = 0.3$ . Compare this result with the analytical solution to your equation.

Answer: numerical 0.40; analytical 0.41

2. A rod of length 1 unit and thermal diffusivity 1 unit is initially at a temperature of zero degrees along its entire length. One end of the rod is then suddenly subjected to a temperature of  $100^\circ$  and is maintained at that value. You may assume that the other end of the rod is perfectly insulated (i.e., there is no temperature gradient at that point). Using two 1D 'rod' elements, and assuming time-stepping parameters  $\Delta t$  and  $\theta$ , set up (but do not solve) the recursive matrix equations that will model the change in temperature along the rod as a function of time.

Answer:  $([M_m] + \theta \Delta t [K_c])\{\Phi\}_1 = ([M_m] - (1 - \theta) \Delta t [K_c])\{\Phi\}_0$

where

$$[M_m] = \frac{1}{12} \begin{bmatrix} 2 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 2 \end{bmatrix} \text{ and } [K_c] = \begin{bmatrix} 2 & -2 & 0 \\ -2 & 4 & -2 \\ 0 & -2 & 2 \end{bmatrix}$$

3. A layer of saturated soil is subjected at time  $t = 0$  to a triangular excess pore pressure distribution varying from 60 units at the top to zero at the bottom. During the subsequent dissipation phase, the top and bottom of the layer can be considered to be fully drained. Using the three-element discretisation shown in Figure 8.36 and a time-stepping scheme with  $\theta = 0.5$ , estimate the pore pressures at the nodes after 0.1 s using a single time step of  $\Delta t = 0.1$ . You may assume the excess pore pressure at the top of the stratum equals zero at all times (including  $t = 0$ ).

Answer: 35.6, 20.9 (consistent); 38.6, 20.0 (lumped); 37.0, 20.0 (lumped-explicit)

4. A rod of length 1 unit and thermal diffusivity 1 unit is initially at a temperature of zero degrees along its entire length. One end of the rod is then subjected to a temperature which increases linearly at a rate of  $1^\circ/\text{s}$  while the other end of the rod is maintained at  $0^\circ$ .

Using two 1D 'rod' elements as shown in Figure 8.37, and assuming a time step of  $\Delta t = 1$  and a weighting factor of  $\theta = 0.5$ , use two steps to estimate the temperature at the central node after 2 s.

Answer:  $T(1) = 0.393$ ,  $T(2) = 0.969$

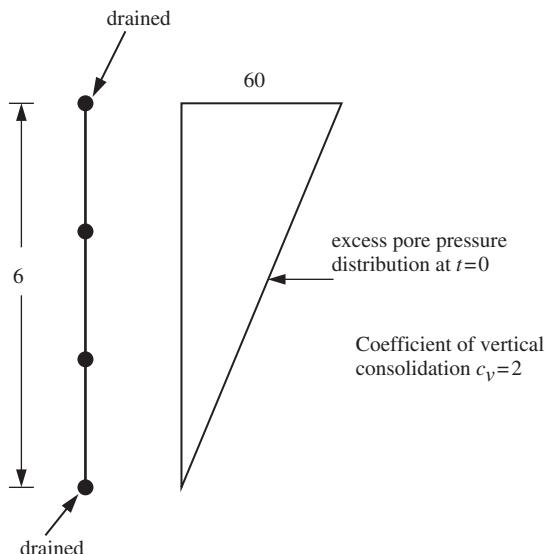


Figure 8.36

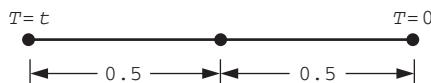


Figure 8.37

5. The rod shown in Figure 8.38 has an initially triangular temperature distribution varying linearly from zero degrees at each end to  $100^\circ$  at the centre. The rod also has a variable diffusivity property as indicated. The rod is allowed to cool while maintaining the ends at zero degrees. Using a single finite element, and a 'lumped mass' discretisation, estimate the temperature after 0.2 s at  $x = 2$  and  $x = 4$  along the rod.

Use one time step of  $\Delta t = 0.2$  with a time-scaling parameter  $\theta = 0.5$ .

Answer: 2-element solution, 47.2, 88.6; 1-element solution, 46.4, 92.8

6. Starting with the governing 2D diffusion equation, go through the Galerkin weighted residual approach to derive terms  $k_{34}$  and  $m_{34}$  of the conductivity and 'mass' matrices of the element shown in Figure 8.39.

Answer:  $k_{34} = c_x b / (6a) - c_y a / (3b)$ ,  $m_{34} = ab / 18$

7. A rod of length  $D$  and thermal diffusivity  $c_x$  is initially at a uniform temperature of  $q_0$  when it is suddenly subjected to a boundary temperature of  $q = 0$  at one end. Assuming the other end of the rod is perfectly insulated, use a single finite element and a lumped 'mass' discretisation to estimate the dimensionless time it will take for the temperature at the insulated end of the rod to cool to one half of its initial value.

Answer:  $T = 0.35$

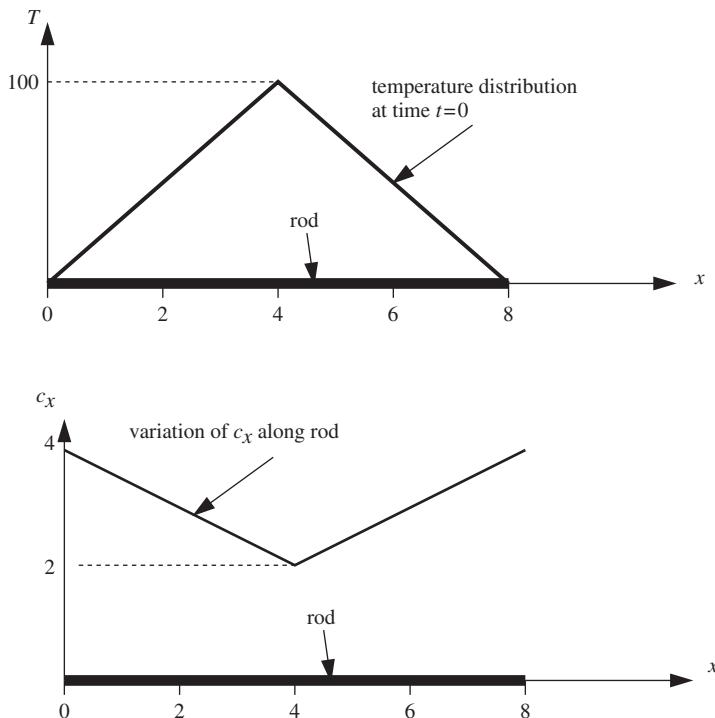


Figure 8.38

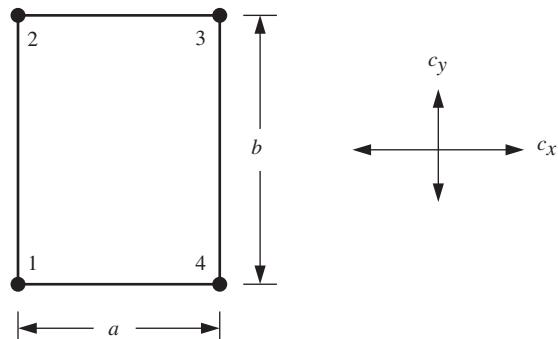


Figure 8.39

8. A rod of unit length and unit thermal diffusivity is at an initial temperature which varies linearly from zero degrees at one end to  $100^\circ$  at the other. The cold end of the rod is perfectly insulated. At time  $t = 0$  the hot end of the rod is suddenly changed to a temperature of zero degrees and maintained at that value. Use a two-element discretisation and an explicit ‘lumped mass’ algorithm to estimate the temperature at the insulated end of the rod after 0.01 and 0.02 s.

Answer:  $T(0.01) = 4.0$ ,  $T(0.02) = 7.4$

9. The rectangular plate shown in Figure 8.40 is initially at zero degrees when the boundaries are suddenly set to  $50^\circ$  and maintained at that value. Select a simple 2D finite element discretisation, and hence estimate the time it takes for the temperature at the centre of the plate to rise to  $25^\circ$ . Use an implicit scheme and consistent ‘mass’ with  $\theta = 0.5$  and  $\Delta t = 0.05$  s.

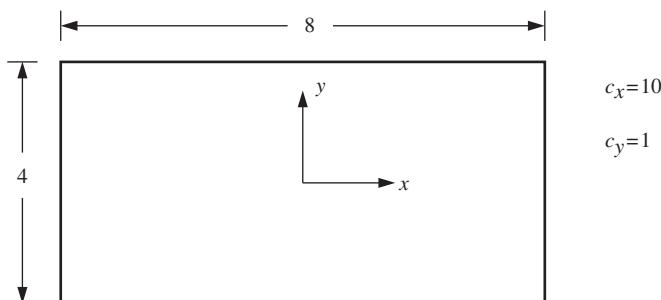


Figure 8.40

Answer:  $t_{50\%} = 0.264$  s

10. A rectangle of saturated soil has an  $x$ -dimension of 4 units and a  $y$ -dimension of 2 units. If drainage is allowed from all four faces of the rectangle and the initial excess pore pressure is set at all points to  $u = 1.0$ , use a single finite element and assume symmetry to estimate the variation of  $u$  with time at the centre of the rectangle if  $c_x = 10$  and  $c_y = 1$ .

Answer: analytical,  $u = e^{-21t/2}$

11. A layer of saturated clay of depth  $D$  and coefficient of consolidation  $c_v$  is drained at its top surface only. The layer is subjected to a sudden excess pore pressure which varies linearly from zero at the surface to  $U_o$  at the base. Using two 1D elements across the soil depth and a dimensionless time step of  $\Delta T = 0.1$ , estimate the average degree of consolidation when  $T = 0.2$ .

Answer: using consistent ‘mass’,  $U = 0.4$

## References

- Carslaw HS and Jaeger JC 1959 *Conduction of Heat in Solids*. Clarendon Press, Oxford.
- Dobbins WE 1944 Effect of turbulence on sedimentation. *Trans Am Soc Civ Eng* **109**, 629–656.
- Huang J and Griffiths DV 2010 One-dimensional consolidation theories for layered soil and coupled and uncoupled solutions by the finite-element method. *Géotechnique* **60**(9), 709–713.
- Incropera FP and DeWitt DP 2002 *Fundamentals of Heat and Mass Transfer*, 5th edn. John Wiley & Sons, Chichester.
- Schiffman RL and Stein JR 1970 One dimensional consolidation of layered systems. *J Soil Mech Found Div, ASCE* **96**(SM4), 1499–1504.
- Smith IM 1976 Integration in time of diffusion and diffusion–convection equations. In *Finite Elements in Water Resources*, Vol. 1. Pentech Press, Plymouth. pp. 3–20.
- Smith IM 1979 The diffusion–convection equation. In *Summary of Numerical Methods for Partial Differential Equations*. Oxford University Press, Oxford, pp. 195–211.
- Smith IM, Farraday RV and O'Connor BA 1973 Rayleigh–Ritz and Galerkin finite elements for diffusion–convection problems. *Water Resour Res* **9**(3), 593–606.
- Taylor DW 1948 *Fundamentals of Soil Mechanics*. John Wiley & Sons, Chichester.

# 9

# Coupled Problems

## 9.1 Introduction

In the previous chapter, flow problems were treated in terms of a single dependent variable, for example the ‘excess pore pressure’ or ‘temperature’, and solutions involved only one degree of freedom per node in the finite element mesh. While this simplification is adequate in some cases, it may be necessary to solve problems in which several degrees of freedom exist at the nodes of the mesh and the several dependent variables, for example velocities and pressures, or displacements and pressures, are ‘coupled’ in the differential equations. Strictly speaking, the equations of two- and three-dimensional elasticity involve coupling between the various components of displacement, but the term ‘coupled problems’ is really reserved for those in which variables of entirely different types are interdependent.

Both steady-state and transient problems are considered in this chapter. As usual, the former involves the solution of sets of simultaneous equations, as in Chapters 4–7. Program 9.1 describes a steady-state solution of the Navier–Stokes equations (see Sections 2.16, 3.11), in which the simultaneous equations are non-linear. An iterative process is therefore necessary during which the equations are solved repeatedly until the velocities and pressures have converged. As discussed in Section 3.11, these equations will have unsymmetrical coefficient matrices.

Program 9.2 solves the same problem without any global matrix assembly using a BiCGStab(l) iterative solver. In this case nested iterative processes are employed, with an internal one for BiCGStab iterations and an external one until convergence of velocities and pressures is obtained. The BiCGStab process is described in Section 3.5.3 (see also Griffiths and Smith, 2006).

The remaining four programs describe coupled transient problems governed by the ‘Biot’ equations (see Sections 2.18, 3.12) (Biot, 1941). These coupled equations are cast as (linear) first-order differential equations in the time variable, and solved by the implicit integration techniques introduced in Chapter 8.

Program 9.3 describes consolidation analysis of a 1D poroelastic material, such as a saturated soil with given drainage boundary conditions, subjected to an incremental time-dependent surface loading (see Section 3.12.2). The next two programs extend this approach to 2D consolidation analysis, using a conventional global assembly and a

direct solver in Program 9.4 and an ‘element-by-element’ approach with a pcg solver in Program 9.5.

The final program in the chapter, Program 9.6, enables investigations to be made of poroelastic–plastic materials and transient collapse problems, by extending Program 9.4 to include non-linear material behaviour governed by a Mohr–Coulomb failure criterion.

### **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 $u-p-v$**

```
PROGRAM p91
!-----
! Program 9.1 Analysis of the plane steady state Navier-Stokes equation
!           using 8-node rectangular quadrilaterals for velocities
!           coupled to 4-node rectangular quadrilaterals for pressures.
!           Mesh numbered in x-direction. Freedoms numbered in the
!           order u-p-v.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::fixed_freedoms,i,iel,itors,k,limit,nband,ndim=2,nels,neq,nip=4, &
nlen,nod=8,nodf=4,nodof=3,nr,ntot=20,nxe,nye
REAL(iwp)::det,one=1.0_iwp,penalty=1.e20_iwp,pt5=0.5_iwp,rho,tol,ubar,    &
vbar,visc,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::g(:,),g_g(:,:,),g_num(:,:,:),nf(:,:,:),no(:, ),node(:, ),      &
num(:, ),sense(:, )
REAL(iwp),ALLOCATABLE::coord(:,:),coordf(:,:),c11(:,:),c12(:,:),c21(:,:),&
c23(:,:),c32(:,:),der(:,:),derf(:,:),deriv(:,:),derivf(:,:),fun(:, ),        &
funf(:, ),g_coord(:,:),jac(:,:),kay(:, ),ke(:, ),loads(:, ),nd1(:,:),          &
nd2(:,:),ndf1(:,:),ndf2(:,:),nfd1(:, ),nfd2(:, ),oldlds(:, ),pb(:, ),       &
points(:,:),uvel(:, ),value(:, ),vvel(:, ),weights(:, ),work(:, ),x_coords(:, ), &
y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,tol,limit,visc,rho
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),uvel(nod),      &
jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),vvel(nod), &
derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),g_g(ntot,nels),            &
c11(nod,nod),c12(nod,nodf),c21(nodf,nod),c23(nodf,nod),g(ntot),        &
c32(nod,nodf),ke(ntot,ntot),fun(nod),x_coords(nxe+1),y_coords(nye+1), &
nf(nodof,nn),g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip),   &
nd1(nod,nod),nd2(nod,nod),nfd1(nod,nodf),nfd2(nod,nodf),nfd1(nodf,nod), &
nfd2(nodf,nod)); READ(10,*)x_coords,y_coords
uvel=zero; vvel=zero; kay=zero; kay(1,1)=visc/rho; kay(2,2)=visc/rho
nf=1; READ(10,*)(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
CALL sample(element,points,weights); nband=0
```

```

!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    g(1:8)=nf(1,num(1:8)); g(9:12)=nf(2,num(1:7:2)); g(13:20)=nf(3,num(1:8))
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
    IF(nband<bandwidth(g))nband=bandwidth(g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
WRITE(11,'(2(A,I5))')                                     &
    " There are",neq," equations and the full bandwidth is",2*(nband+1)-1
ALLOCATE(pb(neq,2*(nband+1)-1),loads(0:neq),oldlds(0:neq),                                     &
    work(nband+1,neq)); loads=zero; oldlds=zero; iters=0
READ(10,*)fixed_freedoms
ALLOCATE(node(fixed_freedoms),sense(fixed_freedoms),                                     &
    value(fixed_freedoms),no(fixed_freedoms))                                     &
READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freedoms)
!-----iteration loop-----
iterations: DO
    iters=iters+1; converged=.FALSE.; pb=zero; work=zero; ke=zero
!-----global matrix assembly-----
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    coordf=coord(1:7:2,:); uvel=(loads(g(1:nod))+oldlds(g(1:nod)))*pt5
    DO i=nod+nodf+1,ntot
        vvel(i-nod-nodf)=(loads(g(i))+oldlds(g(i)))*pt5
    END DO; c11=zero; c12=zero; c21=zero; c23=zero; c32=zero
    gauss_points_1: DO i=1,nip
!-----velocity contribution-----
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der)
       ubar=DOT_PRODUCT(fun,uvel); vbar=DOT_PRODUCT(fun,vvel)
        IF(iters==1)THEN; ubar=one; vbar=zero; END IF
        CALL cross_product(fun,deriv(1,:),nd1)
        CALL cross_product(fun,deriv(2,:),nd2)
        c11=c11+det*weights(i)*(MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)+&
            nd1*ubar+nd2*vbar)
!-----pressure contribution-----
        CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
        jac=MATMUL(derf,coordf); det=determinant(jac); CALL invert(jac)
        derivf=MATMUL(jac,derf)
        CALL cross_product(funf,derivf(1,:),ndf1)
        CALL cross_product(funf,derivf(2,:),ndf2)
        CALL cross_product(funf,deriv(1,:),nfd1)
        CALL cross_product(funf,deriv(2,:),nfd2)
        c12=c12+ndf1*det*weights(i)/rho; c32=c32+ndf2*det*weights(i)/rho
        c21=c21+nfd1*det*weights(i); c23=c23+nfd2*det*weights(i)
    END DO gauss_points_1
    CALL formupv(ke,c11,c12,c21,c23,c32); CALL formtb(pb,ke,g)
END DO elements_2; loads=zero
!-----specify pressure and velocity boundary values-----
DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
pb(no,nband+1)=pb(no,nband+1)+penalty; loads(no)=pb(no,nband+1)*value
!-----equation solution-----
CALL gauss_band(pb,work); CALL solve_band(pb,work,loads); loads(0)=zero
CALL checon(loads,oldlds,tol,converged)

```

```

IF (converged.OR.iters==limit) EXIT
END DO iterations
WRITE(11,'(/A)') " Node      u-velocity  pressure    v-velocity"
DO k=1,nn; WRITE(11,'(I5,A,3E12.4)')k,"      ",loads(nf(:,k)); END DO
WRITE(11,'(/A,I3,A)')" Converged in",iters," iterations."
CALL vecmsh(loads,nf(1:3:2,:),0.3_iwp,0.05_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p91

```

The (steady-state) Navier–Stokes equations were developed in Chapter 2 (Section 2.16). The equilibrium equations to be solved are (2.118), whose coefficients are themselves functions of the velocities  $u$  and  $v$ , so that the equations are non-linear. Furthermore, the coefficient submatrices  $[\mathbf{c}_{ij}]$  are not, in general, symmetrical and reference to Section 3.11 shows that the subroutines `gauss_band` and `solve_band` will be required to operate on the banded equation coefficients.

Section 3.11 illustrates how the element submatrices  $[\mathbf{c}_{ij}]$  are assembled and uses much of the program terminology already developed for uncoupled flow problems in Chapters 7 and 8.

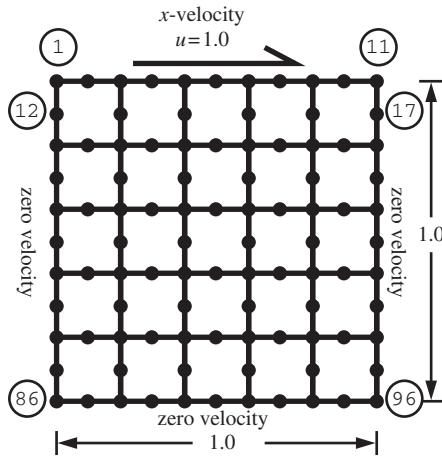
The simple problem chosen to illustrate this program is shown in Figure 9.1. Flow is confined to a rectangular cavity and driven by a uniform horizontal velocity at the top. The velocities at the other three boundaries are set to zero. Eight-node elements are used to model the vector field of velocities, and 4-node elements are used to model the scalar field of pressures. Note that a dummy freedom has been inserted at all mid-side nodes where there is no  $p$  variable. Thus, the second freedom (nodal freedoms are in the order  $u, p, v$ ) at all mid-side nodes is eliminated from the analysis through the ‘restrained freedom’ (`nf`) data.

The first line of data reads the number of elements in each direction of the rectangular mesh (`nxe` and `nye`), the tolerance (`tol`) and iteration ceiling (`limit`) for the non-linear iterations, and the fluid properties given by the molecular viscosity (`visc`) and density (`rho`). The second and third lines give the coordinate data `x_coords` and `y_coords`, and this is followed by the node freedom data which involves `nr=81` restrained nodes in this example. The final lines of data give the fixed velocity boundary condition of  $u = 1.0$  along the 11 top nodes of the mesh.

The structure of the program is described by the structure chart in Figure 9.2. After the data has been read in, various arrays must be initialised to zero. Note that the `kay` matrix which, in the previous chapter, held coefficients of consolidation  $c_x$ , etc., now holds viscosity ( $\mu$ ) and density ( $\rho$ ) in the form  $\mu/\rho$ .

The iteration loop is then entered, controlled by the counter `iters`. System arrays `pb` and `work` must be nulled, together with the element ‘stiffness’ matrix `ke`. Element matrix integration and assembly then proceeds as usual. The nodal coordinates and steering vector are formed as usual by the geometry library subroutine `geom_rect`, with numbering in this case in the  $x$ -direction. Nodal velocities used to form  $\bar{u}$  and  $\bar{v}$  in equation (2.115) are taken to be the average of those computed in the last two iterations. Element submatrices `c11`, etc., are set to zero and the numerical integration loop entered. Average velocities  $\bar{u}$  and  $\bar{v}$  are recovered from `uvel` and `vvel`, except in the first iteration where the ‘guess’  $\bar{u} = 1.0$  and  $\bar{v} = 0.0$  is used.

The submatrix `c11` is formed as required by equations (3.109)–(3.110), followed by submatrices `c12`, `c32`, `c21` and `c23` as demanded by equations (3.111)–(3.112). The element ‘stiffness’ `ke` is built from the submatrices (2.118) by the subroutine `formupv`



```

nx e  ny e  tol  limit  visc  rho
      5      5  0.001   30    0.01   1.0

x_coords, y_coords
0.0   0.2   0.4   0.6   0.8   1.0
0.0  -0.2  -0.4  -0.6  -0.8  -1.0

nr, (k, nf(:,k), i=1, nr)
81
1 1 0 0 2 1 0 0 3 1 1 0 4 1 0 0 5 1 1 0 6 1 0 0
7 1 1 0 8 1 0 0 9 1 1 0 10 1 0 0 11 1 1 0 12 0 0 0
13 1 0 1 14 1 0 1 15 1 0 1 16 1 0 1 17 0 0 0 18 0 1 0
19 1 0 1 21 1 0 1 23 1 0 1 25 1 0 1 27 1 0 1 28 0 1 0
29 0 0 0 30 1 0 1 31 1 0 1 32 1 0 1 33 1 0 1 34 0 0 0
35 0 1 0 36 1 0 1 38 1 0 1 40 1 0 1 42 1 0 1 44 1 0 1
45 0 1 0 46 0 0 0 47 1 0 1 48 1 0 1 49 1 0 1 50 1 0 1
51 0 0 0 52 0 1 0 53 1 0 1 54 1 1 1 55 1 0 1 57 1 0 1
59 1 0 1 61 1 0 1 62 0 1 0 63 0 0 0 64 1 0 1 65 1 0 1
66 1 0 1 67 1 0 1 68 0 0 0 69 0 1 0 70 1 0 1 72 1 0 1
74 1 0 1 76 1 0 1 78 1 0 1 79 0 1 0 80 0 0 0 81 1 0 1
82 1 0 1 83 1 0 1 84 1 0 1 85 0 0 0 86 0 1 0 87 0 0 0
88 0 1 0 89 0 0 0 90 0 1 0 91 0 0 0 92 0 1 0 93 0 0 0
94 0 1 0 95 0 0 0 96 0 1 0

fixed Freedoms, (node(i), value(i), i=1, fixed Freedoms)
11
1 1 1.0 2 1 1.0 3 1 1.0 4 1 1.0 5 1 1.0
6 1 1.0 7 1 1.0 8 1 1.0 9 1 1.0 10 1 1.0
11 1 1.0

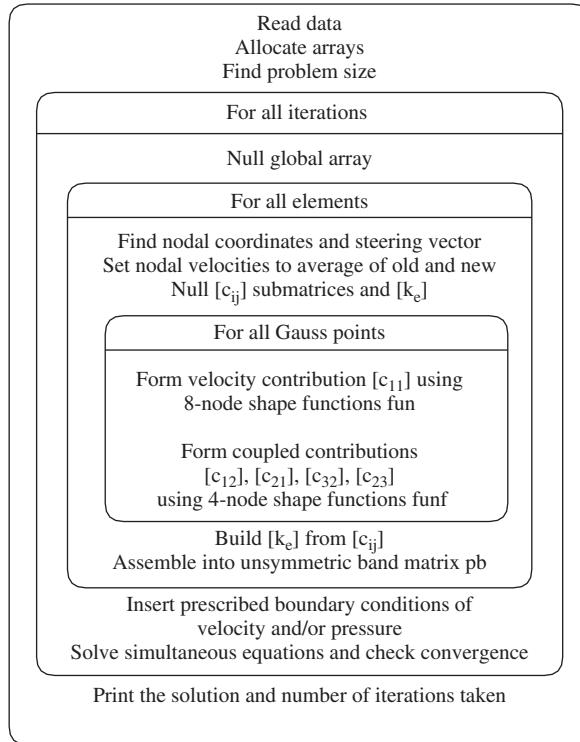
```

**Figure 9.1** Mesh and data for Program 9.1 example

and assembled into the global unsymmetrical band matrix `pb` by the assembly subroutine `formtb`.

It remains only to specify the fixed freedoms by the ‘penalty’ technique (see Section 3.6) and to complete the equation solution using subroutines `gauss_band` and `solve_band`. The maximum number of iterations allowed is 30 (`limit`), but a convergence check of 0.001 (`tol`) is invoked by subroutine `checon`.

The results are listed as Figure 9.3 and velocity vectors generated by subroutine `vecmsh` illustrated in Figure 9.4. Re-gridding strategies (e.g., Kidger, 1994) can further enhance the visualisation.



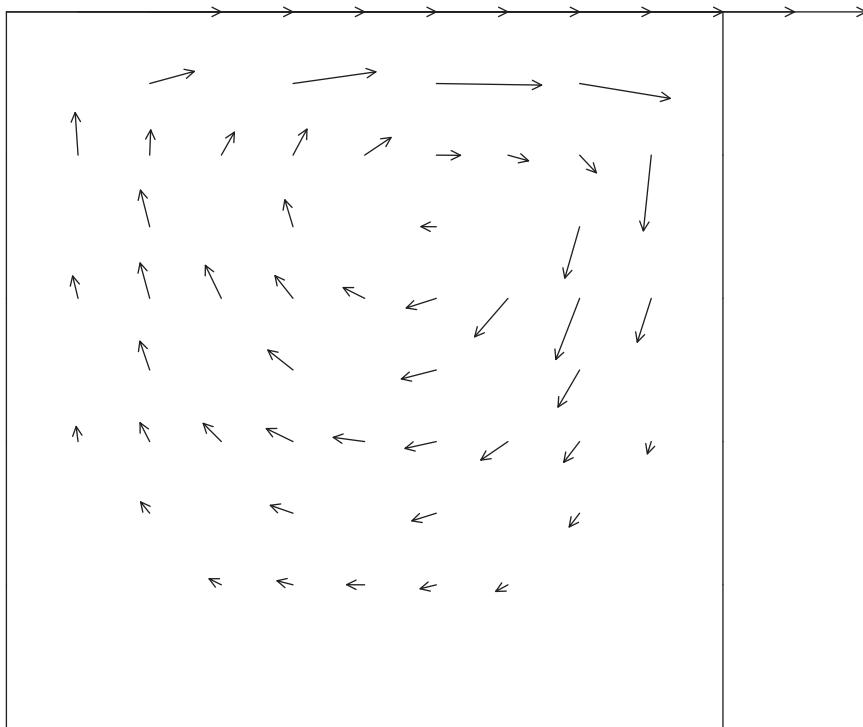
**Figure 9.2** Structure chart for Navier–Stokes analysis with global matrix assembly in Program 9.1

There are 158 equations and the full bandwidth is 79

Node	u-velocity	pressure	v-velocity
1	0.1000E+01	0.0000E+00	0.0000E+00
2	0.1000E+01	0.0000E+00	0.0000E+00
3	0.1000E+01	0.2429E+00	0.0000E+00
4	0.1000E+01	0.0000E+00	0.0000E+00
5	0.1000E+01	0.1704E+00	0.0000E+00
6	0.1000E+01	0.0000E+00	0.0000E+00
7	0.1000E+01	0.2096E+00	0.0000E+00
8	0.1000E+01	0.0000E+00	0.0000E+00
9	0.1000E+01	0.2101E+00	0.0000E+00
10	0.1000E+01	0.0000E+00	0.0000E+00
11	0.1000E+01	0.8023E+00	0.0000E+00
12	0.0000E+00	0.0000E+00	0.0000E+00
13	0.2073E+00	0.0000E+00	0.5662E-01
14	0.3854E+00	0.0000E+00	0.5448E-01
15	0.4901E+00	0.0000E+00	-0.7326E-02
.	.	.	.
90	0.0000E+00	0.2199E+00	0.0000E+00
91	0.0000E+00	0.0000E+00	0.0000E+00
92	0.0000E+00	0.2236E+00	0.0000E+00
93	0.0000E+00	0.0000E+00	0.0000E+00
94	0.0000E+00	0.2275E+00	0.0000E+00
95	0.0000E+00	0.0000E+00	0.0000E+00
96	0.0000E+00	0.2201E+00	0.0000E+00

Converged in 7 iterations.

**Figure 9.3** Results from Program 9.1 example



**Figure 9.4** Velocity vectors at convergence from Program 9.1 example

**Program 9.2 Analysis of the plane steady-state Navier–Stokes equation using 8-node rectangular quadrilaterals for velocities coupled to 4-node rectangular quadrilaterals for pressures. Mesh numbered in  $x$ -direction. Freedoms numbered in the order  $u-p-v$ . Element-by-element solution using BiCGStab(l) with no preconditioning. No global matrix assembly**

```

PROGRAM p92
!-----
! Program 9.2 Analysis of the plane steady state Navier-Stokes equation
! using 8-node rectangular quadrilaterals for velocities
! coupled to 4-node rectangular quadrilaterals for pressures.
! Mesh numbered in x- or y-direction. Freedoms numbered in the
! order u-p-v. Element by element solution using BiCGStab(1).
! with no preconditioning. No global matrix assembly,
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,cg_tot,ell,fixed_freedoms,i,iel,iters,j,k,      &
limit,ndim=2,nels,neq,nip=4,nlen,nn,nod=8,nodf=4,nodof=3,nr,ntot=20,      &
nxe,nye
REAL(iwp)::alpha,beta,cg_tol,det,error,gama,kappa,norm_r,omega,          &

```

```

one=1.0_iwp,penalty=1.e5_iwp,pt5=0.5_iwp,rho,rho1,r0_norm,tol,ubar,      &
vbar,visc,x0,zero=0.0_iwp; LOGICAL::converged,cg_converged
CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::g(:,),g_g(:,:,),g_num(:,:,),nf(:,:,),no(:, ),node(:, ),      &
num(:, ),sense(:, )
REAL(iwp),ALLOCATABLE::b(:, ),coord(:,:, ),coordf(:,:, ),c11(:,:, ),c12(:,:, ),      &
c21(:,:, ),c23(:,:, ),c32(:,:, ),der(:,:, ),derf(:,:, ),deriv(:,:, ),derivf(:,:, ),      &
diag(:,:, ),fun(:,:, ),funf(:,:, ),gamma(:,:, ),gg(:,:, ),g_coord(:,:, ),jac(:,:, ),kay(:,:, ),&
ke(:,:, ),loads(:, ),nd1(:, ),nd2(:, ),nfd1(:, ),nfd2(:, ),ndf1(:, ),      &
nfd2(:, ),oldlds(:, ),points(:, ),r(:, ),rt(:, ),s(:, ),store(:, ),      &
storke(:, ),u(:, ),uvel(:, ),value(:, ),vvel(:, ),weights(:, ),x_coords(:, ),      &
y(:, ),y1(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,tol,limit,visc,rho,cg_tol,cg_limit,x0,e1,kappa
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),      &
jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),      &
derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),nd1(nod,nod),nd2(nod,nod),&
nfd1(nod,nodf),nfd2(nod,nodf),nfd1(nodf,nod),nfd2(nodf,nod),      &
g_g(ntot,nels),c11(nod,nod),c12(nod,nodf),c21(nodf,nod),g(ntot),      &
ke(ntot,ntot),fun(nod),x_coords(nxe+1),y_coords(nye+1),nf(nodof,nn),      &
g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip),c32(nod,nodf),      &
c23(nodf,nod),uvel(nod),vvel(nod),storke(ntot,ntot,nels),s(e1l+1),      &
gg(e1l+1,e1l+1),gamma(e1l+1)); READ(10,*)x_coords,y_coords
uvel=zero; vvel=zero; kay=zero; kay(1,1)=visc/rho; kay(2,2)=visc/rho
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
CALL sample(element,points,weights)
!-----loop the elements to set up global arrays-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    g(:8)=nf(1,num(:8)); g(9:12)=nf(2,num(1:7:2)); g(13:20)=nf(3,num(:8))
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
WRITE(11,'(A,I5,A)')" There are",neq," equations"
ALLOCATE(loads(0:neq),rt(0:neq),r(0:neq,e1l+1),u(0:neq,e1l+1),b(0:neq),      &
diag(0:neq),oldlds(0:neq),y(0:neq),y1(0:neq))
READ(10,*)fixed_freeoms
ALLOCATE(node(fixed_freeoms),sense(fixed_freeoms),      &
value(fixed_freeoms),no(fixed_freeoms),store(fixed_freeoms))
READ(10,*)(node(i),sense(i),value(i),i=1,fixed_freeoms)
iters=0; cg_tot=0; loads=zero; oldlds=zero
!-----iteration loop-----
iterations: do
    iters=iters+1; converged=.FALSE.; ke=zero; diag=zero; b=zero
!-----element stiffness integration and storage -----
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    coordf=coord(1:7:2,:); uvel=(loads(g(:,nod))+oldlds(g(:,nod)))*pt5
    DO i=nod+nodf+1,ntot
        vvel(i-nod-nodf)=(loads(g(i))+oldlds(g(i)))*pt5
    END DO; c11=zero; c12=zero; c21=zero; c23=zero; c32=zero
    gauss_points_1: DO i=1,nip
!-----velocity contribution-----
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    END DO
END DO iterations

```

```

deriv=MATMUL(jac,der)
ubar=DOT_PRODUCT(fun,uvel); vbar=DOT_PRODUCT(fun,vvel)
IF(iters==1)THEN; ubar=one; vbar=zero; END IF
CALL cross_product(fun,deriv(1,:),nd1)
CALL cross_product(fun,deriv(2,:),nd2)
c11=c11+det*weights(i)*(MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)+ &
nd1*ubar+nd2*vbar)
!-----pressure contribution-----
CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
jac=MATMUL(derf,coordf); det=determinant(jac); CALL invert(jac)
derivf=MATMUL(jac,derf)
CALL cross_product(fun,derivf(1,:),ndf1)
CALL cross_product(fun,derivf(2,:),ndf2)
CALL cross_product(funf,deriv(1,:),nfd1)
CALL cross_product(funf,deriv(2,:),nfd2)
c12=c12+nfd1*det*weights(i)/rho; c32=c32+nfd2*det*weights(i)/rho
c21=c21+nfd1*det*weights(i); c23=c23+nfd2*det*weights(i)
END DO gauss_points_1
CALL formupv(ke,c11,c12,c21,c23,c32); storke(:,:,iel)=ke
DO k=1,ntot; diag(g(k))=diag(g(k))+ke(k,k); END DO
END DO elements_2
!-----specify pressure and velocity boundary values-----
DO i=1,fixed_freedoms; no(i)=nf(sense(i),node(i)); END DO
diag(no)=diag(no)+penalty; b(no)=diag(no)*value; store=diag(no)
!---solve the simultaneous equations element by element using BiCGStab(1)-
!-----initialisation phase-----
IF(iters==1)loads=x0; loads(0)=zero; y=loads; y1=zero
elements_3: DO iel=1,nels
  g=g_g(:,:,iel); ke=storke(:,:,:,iel); y1(g)=y1(g)+MATMUL(ke,y(g))
END DO elements_3; cg_iters=0
y1(0)=zero; y1(no)=y(no)*store; y=y1; rt=b-y; r=zeros; r(:,1)=rt; u=zeros
gama=one; omega=one; k=0; norm_r=norm(rt); r0_norm=norm_r; error=one
!-----BiCGStab(1) iterations-----
bicg_iterations: DO
  cg_iters=cg_iters+1; cg_converged=error<cg_tol
  IF(cg_iters==cg_limit.OR.cg_converged)EXIT
  gama=-omega*gama; y=r(:,1)
  DO j=1,ell
    rho1=DOT_PRODUCT(rt,y); beta=rho1/gama
    u(:,:,j)=r(:,:,j)-beta*u(:,:,j); y=u(:,:,j); y1=zeros
    elements_4: DO iel=1,nels
      g=g_g(:,:,iel); ke=storke(:,:,:,iel); y1(g)=y1(g)+MATMUL(ke,y(g))
    END DO elements_4
    y1(0)=zero; y1(no)=y(no)*store; y=y1; u(:,:,j+1)=y
    gama=DOT_PRODUCT(rt,y); alpha=rho1/gama; loads=loads+alpha*u(:,:,1)
    r(:,:,j)=r(:,:,j)-alpha*u(:,:,j+1); y=r(:,:,j); y1=zeros
    elements_5: DO iel=1,nels
      g=g_g(:,:,iel); ke=storke(:,:,:,iel); y1(g)=y1(g)+MATMUL(ke,y(g))
    END DO elements_5; y1(0)=zero; y1(no)=y(no)*store; y=y1; r(:,:,j+1)=y
    END DO; gg=MATMUL(TRANSPOSE(r),r)
    CALL form_s(gg,ell,kappa,omega,gamma,s); loads=loads-MATMUL(r,s)
    r(:,:,1)=MATMUL(r,gamma); u(:,:,1)=MATMUL(u,gamma); norm_r=norm(r(:,:,1))
    error=norm_r/r0_norm; k=k+1
  END DO bicg_iterations; cg_tot=cg_tot+cg_iters
!-----end of BiCGStab(1) process-----
CALL checon(loads,oldlds,tol,converged)
IF(converged.OR.iters==limit)EXIT
END DO iterations

```

```

WRITE(11,'(/A)')" Node      u-velocity  pressure   v-velocity"
DO k=1,nn; WRITE(11,'(I5,A,3E12.4)')k,"      ",loads(nf(:,k)); END DO
WRITE(11,'(/A,I3,A/A,F6.2,A)')" Converged in",iters," iterations",      &
" with an average of", REAL(cg_tot/iters), " BiCGStab(l) iterations."
CALL vecmsh(loads,nf(1:3:2,:),0.3_iwp,0.05_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p92

```

The non-symmetric structures of the element and global matrices generated by Program 9.1 require a different iterative algorithm to the pcg methods used for symmetric equations earlier in the text. In Program 9.2 the BiCGStab(l) algorithm (see Section 3.5.3) is used to re-solve the same cavity flow problem analysed by the previous program. The iterative algorithm requires no global matrix assembly and achieves all global matrix–vector products via gather/scatter algorithms at the element level. Compared with Figure 9.1, the data shown in Figure 9.5 includes an additional line which reads cg\_tol, cg\_limit, x0, ell and kappa described in the notation section at the end of the chapter. The results shown in Figure 9.6 are identical to those in Figure 9.3 using Program 9.1. The output in this case indicates that with ell=4, an average of 31 internal BiCGStab iterations were required for each of the seven external Navier–Stokes iterations.

```

nxe  nye  tol  limit  visc  rho
5      5  0.001  30    0.01  1.0

cg_tol  cg_limit  x0  ell  kappa
1.0e-5  200      1.0  4    0.0

x_coords, y_coords
0.0  0.2  0.4  0.6  0.8  1.0
0.0 -0.2 -0.4 -0.6 -0.8 -1.0

nr, (k,nf(:,k),i=1,nr)
81
1 1 0 0  2 1 0 0  3 1 1 0  4 1 0 0  5 1 1 0  6 1 0 0
7 1 1 0  8 1 0 0  9 1 1 0  10 1 0 0  11 1 1 0  12 0 0 0
13 1 0 1  14 1 0 1  15 1 0 1  16 1 0 1  17 0 0 0  18 0 1 0
19 1 0 1  21 1 0 1  23 1 0 1  25 1 0 1  27 1 0 1  28 0 1 0
29 0 0 0  30 1 0 1  31 1 0 1  32 1 0 1  33 1 0 1  34 0 0 0
35 0 1 0  36 1 0 1  38 1 0 1  40 1 0 1  42 1 0 1  44 1 0 1
45 0 1 0  46 0 0 0  47 1 0 1  48 1 0 1  49 1 0 1  50 1 0 1
51 0 0 0  52 0 1 0  53 1 0 1  54 1 1 1  55 1 0 1  57 1 0 1
59 1 0 1  61 1 0 1  62 0 1 0  63 0 0 0  64 1 0 1  65 1 0 1
66 1 0 1  67 1 0 1  68 0 0 0  69 0 1 0  70 1 0 1  72 1 0 1
74 1 0 1  76 1 0 1  78 1 0 1  79 0 1 0  80 0 0 0  81 1 0 1
82 1 0 1  83 1 0 1  84 1 0 1  85 0 0 0  86 0 1 0  87 0 0 0
88 0 1 0  89 0 0 0  90 0 1 0  91 0 0 0  92 0 1 0  93 0 0 0
94 0 1 0  95 0 0 0  96 0 1 0

fixed Freedoms, (node(i),sense(i),value(i),i=1,fixed Freedoms)
11
1  1  1.0    2  1  1.0    3  1  1.0    4  1  1.0    5  1  1.0
6  1  1.0    7  1  1.0    8  1  1.0    9  1  1.0    10 1  1.0
11 1  1.0

```

**Figure 9.5** Data for Program 9.2 example

There are 158 equations

Node	u-velocity	pressure	v-velocity
1	0.1000E+01	0.0000E+00	0.0000E+00
2	0.1000E+01	0.0000E+00	0.0000E+00
3	0.1000E+01	0.2429E+00	0.0000E+00
4	0.1000E+01	0.0000E+00	0.0000E+00
5	0.1000E+01	0.1704E+00	0.0000E+00
6	0.1000E+01	0.0000E+00	0.0000E+00
7	0.1000E+01	0.2096E+00	0.0000E+00
8	0.1000E+01	0.0000E+00	0.0000E+00
9	0.1000E+01	0.2101E+00	0.0000E+00
10	0.1000E+01	0.0000E+00	0.0000E+00
11	0.1000E+01	0.8023E+00	0.0000E+00
12	0.0000E+00	0.0000E+00	0.0000E+00
13	0.2073E+00	0.0000E+00	0.5662E-01
14	0.3854E+00	0.0000E+00	0.5448E-01
15	0.4901E+00	0.0000E+00	-0.7326E-02
.			
.			
.			
90	0.0000E+00	0.2199E+00	0.0000E+00
91	0.0000E+00	0.0000E+00	0.0000E+00
92	0.0000E+00	0.2236E+00	0.0000E+00
93	0.0000E+00	0.0000E+00	0.0000E+00
94	0.0000E+00	0.2275E+00	0.0000E+00
95	0.0000E+00	0.0000E+00	0.0000E+00
96	0.0000E+00	0.2201E+00	0.0000E+00

Converged in 7 iterations  
with an average of 31.00 BiCGSTAB(1) iterations.

**Figure 9.6** Results from Program 9.2 example

### Program 9.3 One-dimensional coupled consolidation analysis of a Biot poroelastic solid using 2-node ‘line’ elements. Freedoms numbered in the order $v-u_w$

```
PROGRAM p93
!-----
! Program 9.3: One dimensional coupled consolidation analysis of a Biot
!              poro-elastic solid using 2-node "line" elements.
!              Freedoms numbered in order v-uw.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,j,k,loaded_nodes,ndof=4,nels,neq,nlen,nlfp,nls,nn,nod=2,  &
nodof=2,npri,nprops=2,np_types,nr,nres,nstep,ntime
REAL(iwp)::at,a0,dtim,hme,one=1.0_iwp,pt5=0.5_iwp,sett1,theta,time,      &
uav,uavs,zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:,),kdiag(:),nf(:,:,),no(:),num(:)
```

```

REAL(iwp),ALLOCATABLE::al(:,ans(:,c(:,:),ell(:,kc(:,ke(:,&
    km(:,kv(:,lf(:,loads(:,phi0(:,phi1(:,press(:,prop(:,&
    store_kc(:,val(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,np_types; nn=nels+1
ALLOCATE(num(nod),etype(nels),kc(nod,nod),prop(nprops,np_types),
    & ell(nels),nf(nodof,nn),g(ndof),g_g(ndof,nels),km(nod,nod),
    & store_kc(nod,nod,nels),ke(ndof,ndof),phi0(nodof),phi1(ndof),c(nod,nod))
!---prop(1,:) = permeability (k/gamma_w), prop(2,:) = compressibility (m_v)---
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
READ(10,*)dtim,nstep,theta,npri,nres,ntime
nf=1; READ(10,*)nr,(k,nf(:,k),i=1, nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),ans(0:neq),press(nn))
READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes))
READ(10,*)(no(i),val(i),i=1,loaded_nodes)
READ(10,*)nlfp; ALLOCATE(lf(2,nlfp)); READ(10,*)lf
nls=FLOOR(lf(1,nlfp)/dtim); IF(nstep>nls)nstep=nls; ALLOCATE(al(nstep))
CALL load_function(lf,dtim,al); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    num=(/iel,iel+1/); g(1:2)=nf(1,num); g(3:4)=nf(2,num); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')                                     &
    " There are",neq," equations and the skyline storage is",kdiag(neq)
!-----global conductivity and "mass" matrix assembly-----
kv=zero; c(:,1)=-pt5; c(:,2)= pt5
elements_2: DO iel=1,nels
    g=g_g(:,iel); CALL rod_km(kc,prop(1,etype(iel)),ell(iel))
    kc=kc*dtim; CALL rod_km(km,one/prop(2,etype(iel)),ell(iel))
    store_kc(:,iel)=kc; CALL formke(km,kc,c,ke,theta)
    CALL fsparv(kv,ke,g,kdiag)
END DO elements_2; CALL sparin_gauss(kv,kdiag) !---factorise equations---
WRITE(11,'(/2A,I4,A)')" Time           Uav           Uavs",          &
    " Settle_top Pressure (node",nres,")"
loads=zero; hme=zero;
DO iel=1,nels; hme=hme+ell(iel)*prop(2,etype(iel)); END DO;
WRITE(11,'(5E12.4)')0.0,0.0,0.0,0.0,loads(nf(2,nres))
!-----time stepping loop-----
time_steps: DO j=1,nstep; time=j*dtim; ans=zero; a0=zero
    settl=SUM(al(1:j))*hme
    elements_3: DO iel=1,nels
        g=g_g(:,iel); kc=store_kc(:,iel)
        phi0=loads(g(nodof+1:)); phi1=MATMUL(kc,phi0)
        ans(g(nodof+1:))=ans(g(nodof+1:))+phi1
    END DO elements_3
!-----apply loading increment-----
    ans(nf(1,no))=val*al(j)
!-----equation solution-----
    CALL spabac_gauss(kv,ans,kdiag); loads=loads+ans; loads(0)=zero; at=zero
    DO iel=1,nels
        at=at+pt5*ell(iel)*abs(loads(nf(2,iel))+loads(nf(2,iel+1)))
        a0=a0+ell(iel)*SUM(al(1:j))
    END DO
    uav=(a0-at)/a0; uavs=loads(nf(1,1))/settl

```

```

IF(j==ntime)press(:)=loads(nf(2,:))
IF(j/npri*npri==j)WRITE(11,'(5E12.4)')time,uav,uavs,loads(nf(1,1)),      &
loads(nf(2,nres))
END DO time_steps
WRITE(11,'(/A,E10.4,A)')" Depth      Pressure (time=",ntime*dtim,")"
WRITE(11,'(2E12.4)')zero,zero
WRITE(11,'(2E12.4)')(SUM(ell(1:i)),press(i+1),i=1,nels)
STOP
END PROGRAM p93

```

The governing 1D coupled Biot equations to be solved are easily obtained by simplification of the corresponding 2D equations (2.143)–(2.144). If we let  $y$  be the independent spatial variable, with dependent variables  $u_w$  and  $v$  representing the excess pore pressure and displacement (or settlement), respectively, we get

$$\frac{1}{m_v} \frac{\partial^2 v}{\partial y^2} + \frac{\partial u_w}{\partial y} = 0 \quad (9.1)$$

$$\frac{k}{\gamma_w} \frac{\partial^2 u_w}{\partial y^2} + \frac{\partial^2 v}{\partial t \partial y} = 0 \quad (9.2)$$

where  $m_v$  [see (9.6)] and  $k$  are the coefficient of volume compressibility and permeability of the soil, respectively and  $\gamma_w$  is the unit weight of water.

Due to coupling of fluid and solid phases the applied ‘total’ stress  $\sigma$  is divided between a portion carried by the soil skeleton, called ‘effective’ stress  $\sigma'$ , and a portion carried by the pore water, called in soil mechanics the ‘pore pressure’ and denoted in Chapter 2 by  $u_w$ .

After discretisation in space by finite elements, the coupled equations are given by (2.146). These can be seen to be partly algebraic equations and partly first-order differential equations in time. In the incremental load method used here, discretisation in time by the  $\theta$ -method leads to equations (3.121), which are in principle no different from (3.100) for uncoupled problems. If using an assembly approach, solutions will involve setting up the coupled global ‘stiffness’ matrix on the left side of these equations ( $\mathbf{kv}$ ), followed by an equation solution for every time step to obtain the incremental solutions followed by an update of the variables from (3.122). For constant element properties and time step  $\Delta t$ , the left-hand-side matrix  $\mathbf{kv}$  needs to be factorised only once. The remainder of the solution involves matrix-by-vector multiplication on the right-hand-side, which can be done ‘element-by-element’, followed by forward and back-substitution.

Closer examination of equation (3.121) will reveal that some of the diagonal terms of the left-hand-side matrix will be negative, thus the usual Cholesky solution strategy will fail due to the need to take square roots. The subroutine `sparin_gauss`, which uses Gaussian  $[\mathbf{L}][\mathbf{D}][\mathbf{L}]^T$  factorisation, is therefore used for the first time (see Table 3.8).

Inside the time-stepping loop, the right-hand-side vector is summed ‘element-by-element’ from the fluid ‘loading’ and the external load increment held in `a1`. Equation solution is completed using subroutine `spabac_gauss`. Other new subroutines include `load_function`, which reads the input load–time function and linearly interpolates at the resolution of the calculation time step with values held in `a1` and `formke`, which forms the `lhs` element matrix `ke` from equation (3.121).

For a 1D ‘line’ element with two nodes, the nodal freedoms analogous to (2.147) are given by

$$\{u\} = \begin{Bmatrix} v_1 \\ v_2 \end{Bmatrix} \text{ and } \{\mathbf{u}_w\} = \begin{Bmatrix} u_{w1} \\ u_{w2} \end{Bmatrix} \quad (9.3)$$

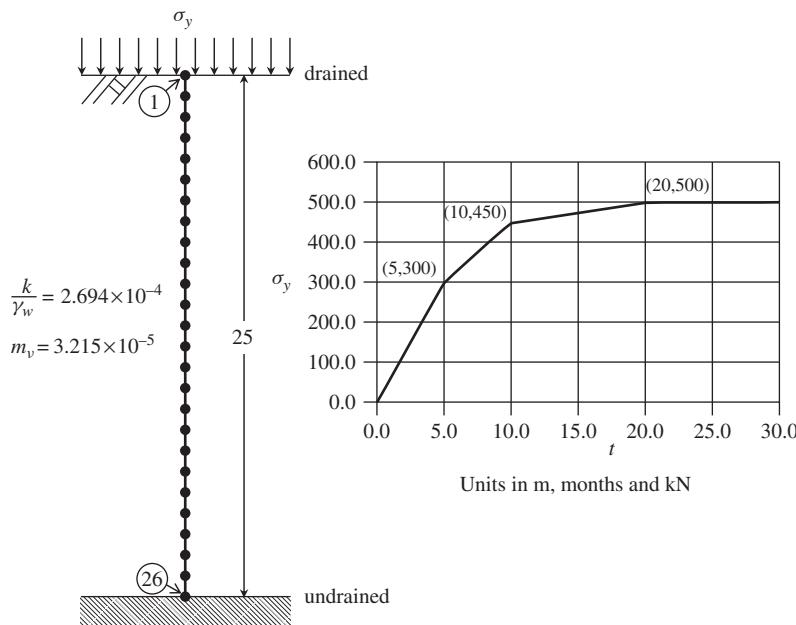


Figure 9.7 Mesh and load-time function for Program 9.3 example

thus each node of the finite element mesh will have two degrees of freedom (in the order  $v, u_w$ ).

The example problem shown in Figure 9.7 is of a 25 m deep layer of saturated clay, drained at the top only, subjected to a time-dependent surface loading. The data file in Figure 9.8 indicates that the layer is subdivided into 25 line elements, each of length 1 m. The soil has uniform properties (`np_types=1`) with a permeability of  $k = 10^{-9}$  m/s [read in as  $k/\gamma_w = 2.694 \times 10^{-4}$  m<sup>4</sup>/(month kN)] and a coefficient of volume compressibility of  $m_v = 3.215 \times 10^{-5}$  m<sup>2</sup>/kN. The time-stepping data calls for 30 steps of 1 month using a  $\theta = 0.5$  implicit scheme. The time history is to be output every 2 months at node 26, and a snapshot of the distribution of excess pore pressure with depth is requested after 10 months. The restrained nodes data with two degrees of freedom per node indicate boundary conditions of zero excess pore pressure (drained) at node 1, and zero settlement (rigid base) at node 26. The loading data shown in Figure 9.7 indicates that one node is to be loaded; that being node 1 with a load weighting of 1.0. The load–time function is defined by five  $(t, \sigma_y)$  coordinates, ramping up in three linear sections as shown in Figure 9.7. The load reaches a maximum value of 500 kPa after 20 months and remains constant thereafter.

The results in Figure 9.9 show that after 30 months, the excess pore pressure at the base of the layer (node 26) has fallen to 283 kPa having reached a maximum of 434 kPa after 10 months, the settlement is 0.26 m, and the average degree of consolidation is 64%. The ultimate consolidation settlement will therefore be 0.40 m, which is to be expected from 1D settlement theory. It may be noted that the average degree of consolidation is the same whether computed using excess pore pressures or settlement, because  $m_v$  is constant throughout the layer (see also Program 8.2 and Huang and Griffiths, 2010).

```

nels  np_types
25      1

prop (k/gamma, m_v)
2.694E-4  3.215E-5

etype (not needed)

ell
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
1.0  1.0  1.0  1.0  1.0

dtim  nstep  theta  npri  nres  ntime
1.0    30      0.5     2      26     10

nr, (k,nf(:,k),i=1,nr)
2
1 1 0 26 0 1

loaded_nodes, (no(i),val(i),i=1,loaded_nodes)
1
1 1.0

nlfp, (lf(:,i),i=1,nlfp)
5
0.0  0.0  5.0 300.0
10.0 450.0 20.0 500.0
30.0 500.0

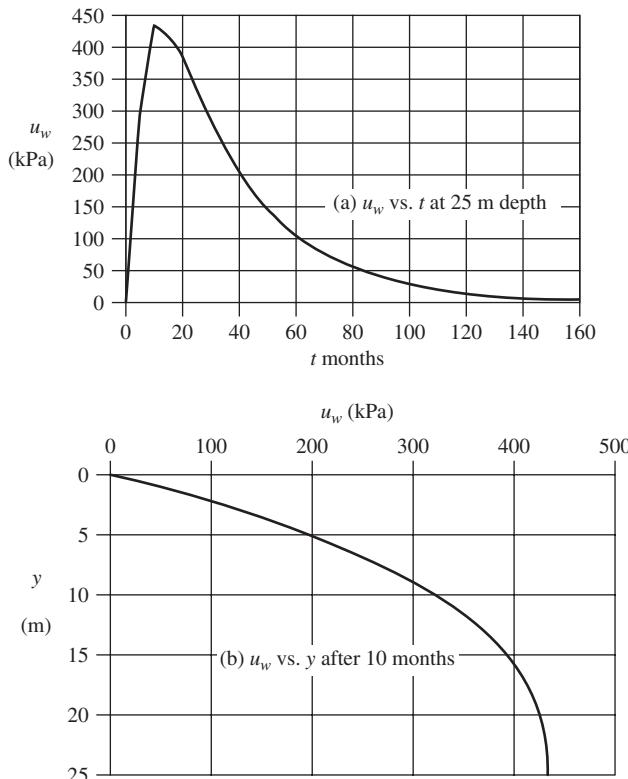
```

**Figure 9.8** Data for Program 9.3 example

There are 50 equations and the skyline storage is 170

Time	Uav	Uavs	Settle_top	Pressure (node 26)
0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.2000E+01	0.1228E+00	0.1228E+00	0.1185E-01	0.1200E+03
0.4000E+01	0.1740E+00	0.1740E+00	0.3356E-01	0.2398E+03
0.6000E+01	0.2252E+00	0.2252E+00	0.5972E-01	0.3282E+03
0.8000E+01	0.2685E+00	0.2685E+00	0.8415E-01	0.3833E+03
0.1000E+02	0.3023E+00	0.3023E+00	0.1094E+00	0.4337E+03
0.1200E+02	0.3537E+00	0.3537E+00	0.1308E+00	0.4295E+03
.	.	.	.	.
0.2600E+02	0.5870E+00	0.5870E+00	0.2359E+00	0.3225E+03
0.2800E+02	0.6137E+00	0.6137E+00	0.2466E+00	0.3024E+03
0.3000E+02	0.6386E+00	0.6386E+00	0.2566E+00	0.2833E+03
.	.	.	.	.
Depth	Pressure (time=0.1000E+02)			
0.0000E+00	0.0000E+00			
0.1000E+01	0.4523E+02			
0.2000E+01	0.8876E+02			
.	.	.	.	.
0.2300E+02	0.4322E+03			
0.2400E+02	0.4333E+03			

**Figure 9.9** Results from Program 9.3 example



**Figure 9.10** Plotted results from Program 9.3 example

The plotted results in Figure 9.10 show (a) the excess pore pressure at the base of the layer vs. time (up to 160 months) and (b) the excess pore pressure vs. depth (after 10 months).

#### 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

```
PROGRAM p94
!-----
! 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-uw. Incremental load version.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,j,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nlen,nlfp, &
nls,nn,nod=8,nodf=4,nodof=3,npri,nprops=4,np_types,nr,nres,nst=3,nstep, &
ntot=20,nxe,nye; CHARACTER(LEN=15)::argv,element='quadrilateral'
```

```

REAL(iwp)::det,dtim,theta,time,tot_load,zero=0.0_iwp
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_(:, :, ),g_num(:, :, ),kdiag(:,nf(:, :, ), &
no(:, ),num(:, )
REAL(iwp),ALLOCATABLE::al(:,ans(:,bee(:, :, ),c(:, :, ),coord(:, :, ),dee(:, :, ), &
der(:, :, ),derf(:, :, ),deriv(:, :, ),derivf(:, :, ),eld(:, ),fun(:, ),funf(:, ),gc(:, ), &
g_coord(:, :, ),jac(:, :, ),kay(:, :, ),ke(:, :, ),km(:, :, ),kc(:, :, ),kv(:, ),lf(:, :, ), &
loads(:, ),phi0(:, ),phil(:, ),points(:, :, ),prop(:, :, ),sigma(:, ), &
store_kc(:, :, :, ),val(:, :, ),vol(:, ),volf(:, :, ),weights(:, ),x_coords(:, ), &
y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(dee(nst,nst),points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),&
jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod), &
derf(ndim,nodf),funf(ndof),bee(ndof,ndof),km(ndof,ndof),kc(ndof,nodf), &
g_g(ntot,nels),ke(ntot,ntot),c(ndof,nodf),x_coords(nxe+1), &
y_coords(nye+1),vol(ndof),nf(ndof,nn),g(ntot),volf(ndof,nodf), &
g_coord(ndim,nn),g_num(nod,nels),num(nod),weights(nip), &
store_kc(ndof,nodf,nels),phi0(ndof),phil(ndof),prop(nprops,np_types), &
etype(nels),eld(ndof),gc(ndim),sigma(nst),fun(nod))
READ(10,*)prop; etype=1; if(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),ans(0:neq))
READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(no(i),val(i,:),i=1,loaded_nodes)
READ(10,*)nlfp; ALLOCATE(lf(2,nlfp))
READ(10,*)lf; nls=FLOOR(lf(1,nlfp)/dtim); IF(nstep>nls)nstep=nls
ALLOCATE(al(nstep)); CALL load_function(lf,dtim,al); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    g(1:15:2)=nf(1,num(:)); g(2:16:2)=nf(2,num(:)); g(17:)=nf(3,num(1:7:2))
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
    CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')
    " There are",neq," equations and the skyline storage is",kdiag(neq)
CALL sample(element,points,weights); loads=zero; kv=zero
!-----global matrix assembly-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel))); num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero; c=zero; kc=zero
gauss_points_1: DO i=1,nip
!-----elastic solid contribution-----
    CALL shape_der(der,points,i); jac=MATMUL(der,coord)
    det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
    CALL beemat(bee,deriv); vol(:)=bee(1,:)+bee(2,:)
    km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----fluid contribution-----
    CALL shape_fun(funf,points,i); CALL shape_der(darf,points,i)
    derivf=MATMUL(jac,derf)
    kc=kc+MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtim
    CALL cross_product(vol,funf,volf); c=c+volf*det*weights(i)

```

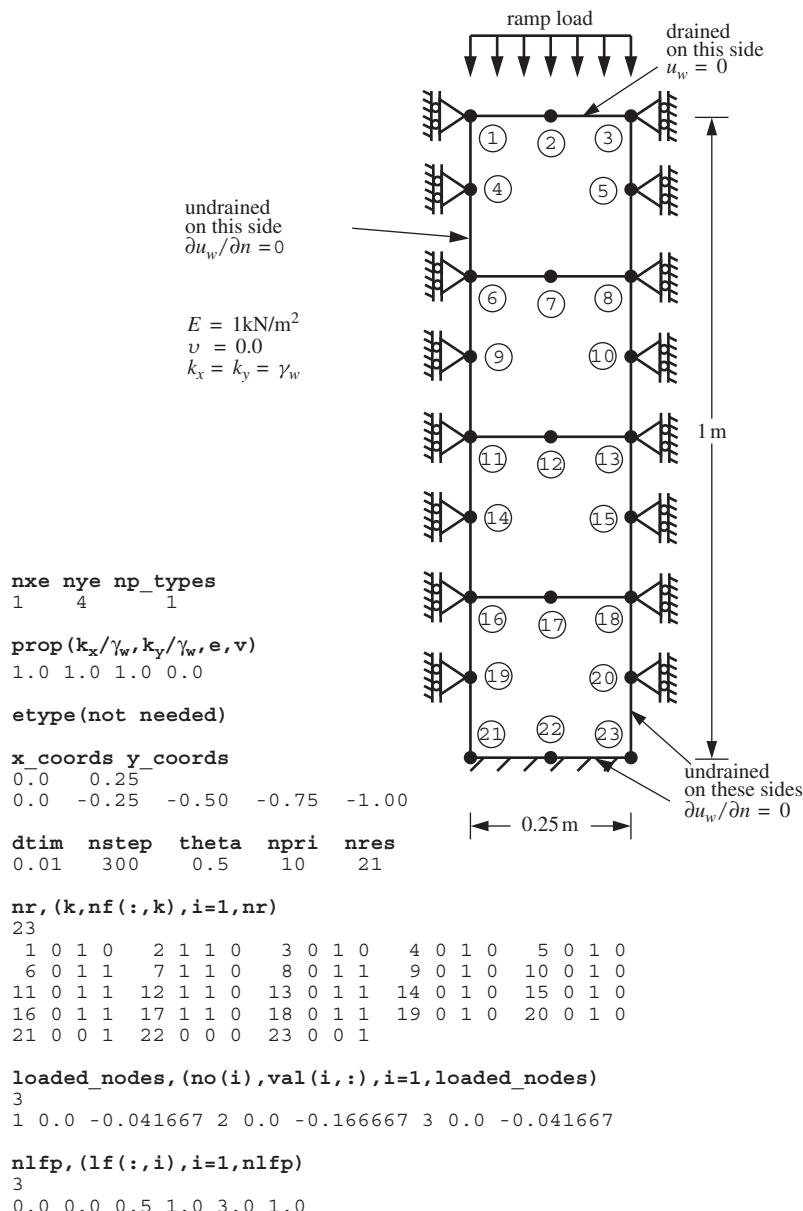
```

END DO gauss_points_1; store_kc(:,:,iel)=kc
CALL formke(km,kc,c,ke,theta); CALL fsparv(kv,ke,g,kdiag)
END DO elements_2; CALL sparin_gauss(kv,kdiag) !---factorise equations---
!-----time stepping loop-----
WRITE(11,'(/A,I5)')" Results at node",nres
WRITE(11,'(A)')                                     &
"      time          load       x-disp       y-disp     porepressure"
WRITE(11,'(5E12.4)')0.0,0.0,loads(nf(:,nres))
time_steps: DO j=1,nstep
    tot_load=SUM(al(1:j)); time=j*dtim; ans=zero
elements_3: DO iel=1,nels
    g=g_g(:,:,iel); kc=store_kc(:,:,iel)
    phi0=loads(g(ndof+1:)); phi1=MATMUL(kc,phi0)
    ans(g(ndof+1:))=ans(g(ndof+1:))+phi1
END DO elements_3
!-----apply loading increment-----
DO i=1,loaded_nodes; ans(nf(1:2,no(i)))=val(i,:)*al(j); END DO
!-----equation solution-----
CALL spabac_gauss(kv,ans,kdiag); loads=loads+ans; loads(0)=zero
IF(j/npri*npri==j)WRITE(11,'(5E12.4)')time,tot_load,loads(nf(:,nres))
!-----recover stresses at nip integrating points-----
!  nip=1; DEALLOCATE(points,weights)
!  ALLOCATE(points(nip,ndim),weights(nip))
!  CALL sample(element,points,weights)
!  WRITE(11,'(A,I2,A)')" The integration point (nip=",nip,") stresses are:"
!  WRITE(11,'(A,A)')" Element x-coord      y-coord",           &
!                  "      sig_x      sig_y      tau_xy"
!  elements_4: DO iel=1,nels; num=g_num(:,:,iel)
!    coord=TRANSPOSE(g_coord(:,:,num)); g=g_g(:,:,iel); eld=loads(g(:ndof))
!    CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
!    gauss_pts_2: DO i=1,nip
!      CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
!      gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
!      deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
!      sigma=MATMUL(dee,MATMUL(bee,eld))
!      IF(j/npri*npri==j)WRITE(11,'(I5,6E12.4)')iel,gc,sigma
!    END DO gauss_pts_2
!  END DO elements_4
END DO time_steps
CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p94

```

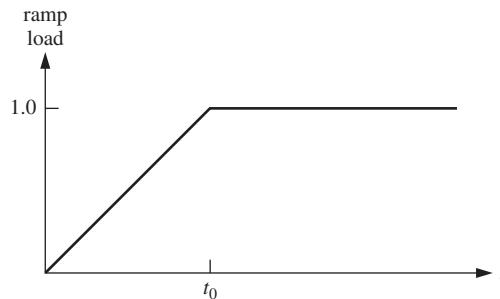
The extension of Program 9.3 to two dimensions results in each node having three degrees of freedom, that is, two components of displacement and an excess pore pressure (in the order  $u$ ,  $v$  and  $u_w$ ).

The problem chosen is of a plane strain ‘oedometer’ specimen as shown by the mesh and input data given in Figure 9.11. The base and sides of the mesh are impermeable ‘no-flow’ boundaries, and ‘smooth’ roller boundary conditions are imposed on the sides. The top of the specimen is drained, and subjected to the ‘ramp’ loading shown in Figure 9.12 which indicates a linearly increasing load reaching a maximum of 1.0 at time  $t_o$ .



**Figure 9.11** Mesh and data for Program 9.4 example

The first line of data reads the number of elements in each direction of the rectangular mesh (`nxe` and `nye`), and the number of property types (`np_types`). The properties are read next in the order  $k_x/\gamma_w$ ,  $k_y/\gamma_w$ ,  $E'$  and  $\nu'$ , where  $k_x$  and  $k_y$  are the permeabilities in the  $x$ - and  $y$ -directions,  $\gamma_w$  is the unit weight of water, and  $E'$  and  $\nu'$  are the effective Young's modulus and Poisson's ratio of the soil matrix. Since `np_types=1`



**Figure 9.12** Ramp loading

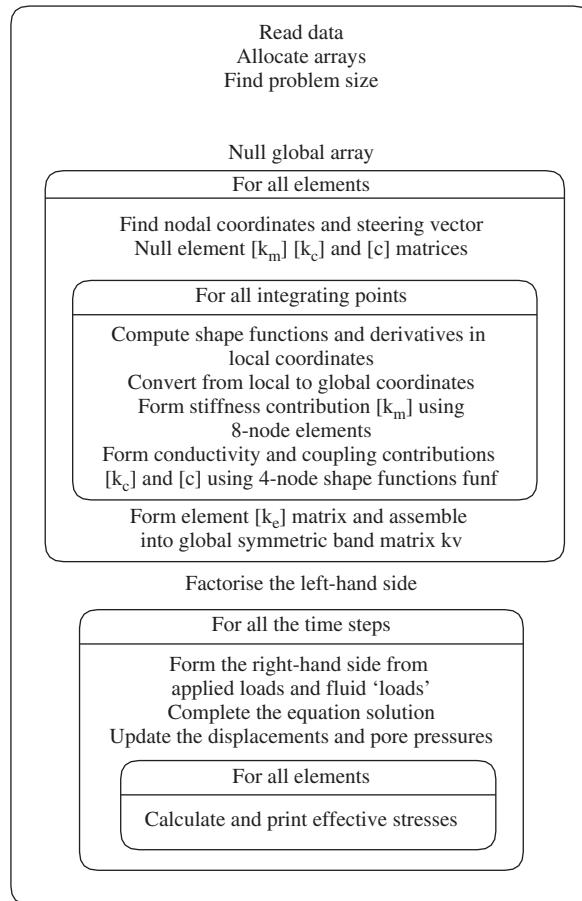
in this homogeneous example, the `etype` data is not needed. Next comes the rectangular mesh coordinate data `x_coords` and `y_coords`, followed by the time-stepping and output data. In this case, the data calls for `nstep=300` calculation steps, with a time step of `dtime=0.01`. The time-stepping parameter is set to `theta=0.5`. Displacement and pore pressure output is requested at node `nres=21` at every tenth time step `npri=10`. The nodal freedom data comes next, fixing the pore pressures at the top of the mesh to zero (drained), and also the  $x$ -displacements at each side of the mesh to zero (smooth). This implies ‘oedometer’ conditions, with drainage at the top only. As with velocities in Programs 9.1 and 9.2, no pressures are computed at the mid-side nodes, so the third freedom at all mid-side nodes is removed from the analysis. The next data provide the load weightings corresponding to a unit pressure (Appendix A) applied to the three nodes at the top of the mesh. The final two lines of data provide the three points that define the load–time function to be applied at the top of the mesh.

Turning to the program, the nodal coordinates and steering vector are again provided by subroutine `geom_rect` with numbering in the  $x$ -direction. Subroutine `formke` builds up the `ke` matrix from constituent matrices `km`, `kc` and `c`, which are obtained from (3.66), (3.69) and (3.117) by numerical integration, before assembly into the (symmetric) global skyline matrix `kv`. The structure chart for this program is given in Figure 9.13.

As in Program 9.3, the subroutine `sparin_gauss` is used to factorise the symmetric left-hand-side global matrix `kv` and the time-stepping loop is entered. The right-hand-side vector is summed ‘element-by-element’ from the fluid ‘loading’ and the external load increment held in `a1`. The subroutine `spabac_gauss` completes the solution and the element effective stresses can be recovered if required at the element Gauss points (in the current version this output has been commented out).

The results are shown as Figure 9.14, and the pore pressure at the base of the mesh (node 21) is plotted against time in Figure 9.15 for two different ramp rise times,  $T_0 = 0.1$  and  $T_0 = 0.5$ . The dimensionless ‘time factor’  $T$  is defined as

$$T = \frac{c_v t}{D^2} \quad (9.4)$$



**Figure 9.13** Structure chart for incremental form of Biot analysis with global matrix assembly as used in Program 9.4

where  $D$  is the ‘maximum drainage path’. The coefficient of consolidation  $c_v$  is found from

$$c_v = \frac{k_v}{m_v \gamma_w} \quad (9.5)$$

where

$$m_v = \frac{(1 + v')(1 - 2v')}{E'(1 - v')} \quad (9.6)$$

and  $k_v = k_y$  is the soil permeability in the vertical direction. In the present example,  $v' = 0$  and  $E' = 1.0$ , so  $m_v = 1.0$ . In addition,  $D = 1.0$  and  $k_v/\gamma_w = 1.0$ , so  $T = t$ . The results are in very close agreement with Schiffman (1960), and problems of practical importance have been solved since (Smith and Hobbs, 1976).

There are 32 equations and the skyline storage is 280

Results at node 21

time	load	x-disp	y-disp	porepressure
0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00
0.1000E+00	0.2000E+00	0.0000E+00	0.0000E+00	-0.1994E+00
0.2000E+00	0.4000E+00	0.0000E+00	0.0000E+00	-0.3743E+00
0.3000E+00	0.6000E+00	0.0000E+00	0.0000E+00	-0.5125E+00
0.4000E+00	0.8000E+00	0.0000E+00	0.0000E+00	-0.6203E+00
0.5000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.7043E+00
0.6000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.5703E+00
0.7000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.4463E+00
0.8000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.3478E+00
0.9000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.2709E+00
0.1000E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2110E+00
0.1100E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1643E+00
0.1200E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1280E+00
0.1300E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.9968E-01
0.1400E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.7764E-01
0.1500E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.6047E-01
0.1600E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.4709E-01
0.1700E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.3668E-01
0.1800E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2857E-01
0.1900E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2225E-01
0.2000E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1733E-01
0.2100E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1350E-01
0.2200E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1051E-01
0.2300E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.8187E-02
0.2400E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.6377E-02
0.2500E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.4966E-02
0.2600E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.3868E-02
0.2700E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.3013E-02
0.2800E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2346E-02
0.2900E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1827E-02
0.3000E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1423E-02

Figure 9.14 Results from Program 9.4 example

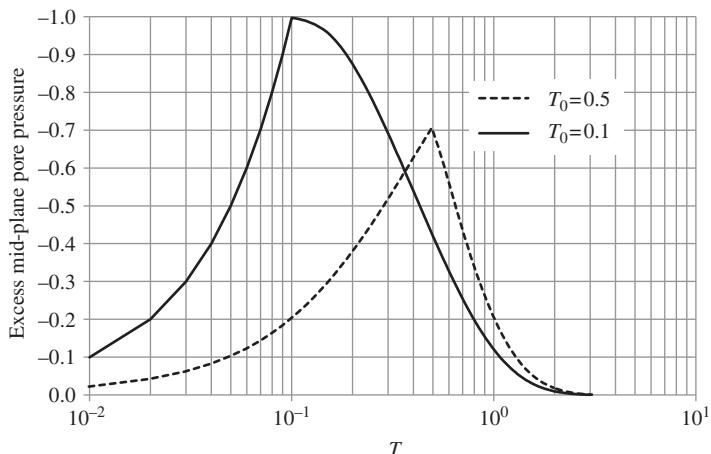


Figure 9.15 Mid-plane pore pressure response to ramp loading from Program 9.4 example

**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**

```

PROGRAM p95
!-----
! Program 9.5 Plane strain consolidation analysis of a Biot elastic solid
!           using 8-node rectangular quadrilaterals for displacements
!           coupled to 4-node rectangular quadrilaterals for pressures.
!           Freedoms numbered in order u-v-uw. Incremental load version.
!           No global stiffness matrix assembly.
!           Diagonally preconditioned conjugate gradient solver.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,i,iel,j,k,loaded_nodes,ndim=2,ndof=16,nels,    &
neq,nip=4,nlen,nlfp,nls,nn,nod=8,nodf=4,nodof=3,npri,nprops=4,np_types,&
nr,nres,nst=3,nstep,ntot=20,nxe,nye; LOGICAL::cg_converged
REAL(iwp)::alpha,beta,cg_tol,det,dtim,one=1.0_iwp,theta,time,tot_load,up,&
zero=0.0_iwp; CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:,g_g(:,:,)),g_num(:,:,),nf(:,:,),no(:, ),&
num(:)
REAL(iwp),ALLOCATABLE::al(:,ans(:,bee(:,:,c(:,:))),coord(:,:,d(:)),&
dee(:,:,der(:,:,derf(:,:,deriv(:,:,derivf(:,:,diag_precon(:,:,&
eld(:,:,fun(:),funf(:),gc(:),g_coord(:,:,jac(:,:,kay(:,:,ke(:,:,&
km(:,:,kc(:,:,lf(:,:,loads(:,p(:),points(:,:,prop(:,:,sigma(:),&
storke(:,:,u(:),val(:,:,vol(:),volf(:,:,store_kc(:,:,&
weights(:,x(:,xnew(:,x_coords(:,y_coords(:,phi0(:,phi1(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(dee(nst,nst),points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),&
jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),&
derf(ndim,nodf),funf(ndof),bee(nst,ndof),km(ndof,ndof),kc(ndof,nodf),&
g_g(ntot,nels),ke(ntot,ntot),c(ndof,nodf),fun(nod),x_coords(nxe+1),&
store_kc(ndof,nodf,nels),y_coords(nye+1),vol(ndof),nf(nodof,nn),&
g(ntot),volf(ndof,nodf),g_coord(ndim,nn),g_num(nod,nels),num(nod),&
weights(nip),storke(ntot,ntot,nels),etype(nels),phi0(nodf),phi1(nodf),&
prop(nprops,np_types),gc(ndim),sigma(nst),eld(ndof))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords; READ(10,*)dtim,nstep,theta,npri,nres
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(loads(0:neq),ans(0:neq),p(0:neq),x(0:neq),xnew(0:neq),u(0:neq), &
diag_precon(0:neq),d(0:neq))
READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(no(i),val(i,:),i=1,loaded_nodes)

```

```

READ(10,*)nlfp; ALLOCATE(lf(2,nlfp)); READ(10,*)lf
nls=FLOOR(lf(1, nlfp)/dtime); IF(nstep>nls)nstep=nls
ALLOCATE(al(nstep)); CALL load_function(lf,dtime,al)
!-----loop the elements to set up element data-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    g(1:15:2)=nf(1,num(:)); g(2:16:2)=nf(2,num(:))
    g(17:20)=nf(3,num(1:7:2)); g_g(:,iel)=g
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
WRITE(11,'(A,I5,A)')" There are",neq," equations"
loads=zero; p=zero; xnew=zero; diag_precon=zero
CALL sample(element,points,weights)
!-----element matrix integration, storage and preconditioner-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    km=zero; c=zero; kc=zero
    gauss_points_1: DO i=1,nip
!-----elastic solid contribution-----
        CALL shape_der(der,points,i); jac=MATMUL(der,coord)
        det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
        CALL beemat(bee,deriv); vol(:)=bee(1,:)+bee(2,:)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----fluid contribution-----
        CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
        derivf=MATMUL(jac,derf)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtime
        CALL cross_product(vol,funf,volf); c=c+volf*det*weights(i)
    END DO gauss_points_1
    store_kc(:,:,iel)=kc; CALL formke(km,kc,c,ke,theta); storke(:,:,iel)=ke
    DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+theta*km(k,k); END DO
    DO k=1,ndof
        diag_precon(g(ndof+k))=diag_precon(g(ndof+k))-theta*theta*kc(k,k)
    END DO
    END DO elements_2
    diag_precon(1)=one/diag_precon(1); diag_precon(0)=zero
!-----time stepping loop-----
    WRITE(11,'(/A,I5)')" Results at node",nres
    WRITE(11,'(4X,A)')                                     &
        "time          load         x-disp       y-disp      porepressure   cg iters"
    WRITE(11,'(5E12.4)')0.0,0.0,loads(nf(:,nres))
    time_steps: DO j=1,nstep; tot_load=SUM(al(1:j)); time=j*dtime; ans=zero
    elements_3: DO iel=1,nels
        g=g_g(:,iel); kc=store_kc(:,:,iel); phi0=loads(g(ndof+1:)) ! gather
        phil=MATMUL(kc,phi0); ans(g(ndof+1:))=ans(g(ndof+1:))+phil
    END DO elements_3; ans(0)=zero
    DO i=1,loaded_nodes; ans(nf(1:2,no(i)))=val(i,:)*al(j); END DO
    d=diag_precon*ans; p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
    pcg: DO
        cg_iters=cg_iters+1; u=zero
    elements_4: DO iel=1,nels

```

```

    g=g_g(:,iel); ke=storke(:,:,iel); u(g)=u(g)+MATMUL(ke,p(g))
  END DO elements_4
  up=DOT_PRODUCT(ans,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
  ans=ans-u*alpha; d=diag_precon*ans; beta=DOT_PRODUCT(ans,d)/up
  p=d+p*beta; CALL checon(xnew,x,cg_tol,cg_converged)
  IF(cg_converged.OR.cg_iters==cg_limit)EXIT
  END DO pcg; ans=xnew; ans(0)=zero; loads=loads+ans; loads(0)=zero
  IF(j/npri*npri==j)WRITE(11,'(5E12.4,I7)')
    time,tot_load,loads(nf(:,nres)),cg_iters
  &
!-----recover stresses at nip integrating points-----
!  nip=1; DEALLOCATE(points,weights)
!  ALLOCATE(points(nip,ndim),weights(nip))
!  CALL sample(element,points,weights)
!  WRITE(11,'(A,I2,A)')" The integration point (nip=",nip,") stresses are:"
!  WRITE(11,'(A,A)')" Element x-coord      y-coord",
!    "      sig_x      sig_y      tau_xy"
!  elements_5: DO iel=1,nels; num=g_num(:,iel);
!    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); eld=loads(g(:ndof))
!    CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
!    gauss_pts_2: DO i=1,nip
!      CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
!      gc=MATMUL(fun,coord); jac=MATMUL(der,coord); CALL invert(jac)
!      deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
!      sigma=MATMUL(dee,MATMUL(bee,eld))
!      IF(j/npri*npri==j)WRITE(11,'(I5,6E12.4)')iel,gc,sigma
!    END DO gauss_pts_2
!  END DO elements_5
  END DO time_steps
  CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
  CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p95

```

In this program, we repeat the example described in Program 9.4, but this time using an ‘element-by-element’ approach with a pcg solver that involves no global matrix assembly. The processes given by (3.121) can all be performed at the element level using the pcg algorithm. As in Program 9.4, subroutine `formke` forms the (symmetric)  $[\mathbf{k}_e]$  matrix called (`ke`) on the left-hand side. Inspection of this matrix shows that  $[\mathbf{k}_e]$  is not positive definite due to negative terms associated with  $[\mathbf{k}_c]$ . Simple diagonal preconditioning does yield a symmetric positive definite preconditioned matrix, however, and this is the method used in Program 9.5. It is recognised that alternative preconditioning and iterative strategies may well be more efficient. The `ke` matrices are stored in `storke`. Inside the time-stepping loop, the right-hand-side vector is summed element-by-element from the fluid ‘loading’ and the external load increment held in `al`.

The data are listed as Figure 9.16 and the extra data items are just the conjugate gradient iteration tolerance and iteration limit, `cg_tol` and `cg_limit`, respectively. The results are listed as Figure 9.17, which are essentially identical to those in Figure 9.14, and indicate that approximately 13 pcg iterations were needed on average at each calculation time step.

```

nxr  nyx  cg_tol   cg_limit  np_types
1      4    1.0e-5     200      1

prop(k_x/g_w,k_y/g_w,e,v)
1.0  1.0  1.0  0.0

etype(not needed)

x_coords y_coords
0.0    0.25
0.0   -0.25  -0.50  -0.75  -1.00

dtim  nstep  theta  npri  nres
0.01  300    0.5    10    21

nr,(k,nf(:,k),i=1,nr)
23
1 0 1 0  2 1 1 0  3 0 1 0  4 0 1 0  5 0 1 0
6 0 1 1  7 1 1 0  8 0 1 1  9 0 1 0  10 0 1 0
11 0 1 1  12 1 1 0  13 0 1 1  14 0 1 0  15 0 1 0
16 0 1 1  17 1 1 0  18 0 1 1  19 0 1 0  20 0 1 0
21 0 0 1  22 0 0 0  23 0 0 1

loaded_nodes, (no(i),val(i,:),i=1,loaded_nodes)
3
1 0.0  -0.041667  2 0.0  -0.166667  3 0.0  -0.041667

nlfp,(lf(:,i),i=1,nlfp)
3
0.0 0.0  0.5 1.0  3.0 1.0

```

**Figure 9.16** Data for Program 9.5 example

**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**

```

PROGRAM p96
!-----
! Program 9.6 Plane strain consolidation analysis of a Biot elastic-plastic
!           (Mohr-Coulomb) material using 8-node rectangular
!           quadrilaterals for displacements coupled to 4-node
!           rectangular quadrilaterals for pressures.
!           Freedoms numbered in order u-v-uw. Viscoplastic strain method.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,itors,j,k,limit,loaded_nodes,ndim=2,ndof=16,nels,neq,      &
          nip=4,nlen,nlfp,nls,nn,nod=8,nodf=4,nodof=3,npri,nprops=7,np_types,  &
          nr,nres,nst=4,nstep,ntot=20,nxe,nye
REAL(iwp)::coh,cons,ddt,det,dpore,dq1,dq2,dq3,dsbar,dt,dtim,d4=4.0_iwp, &
           d180=180.0_iwp,e,f,lode_theta,one=1.0_iwp,phi,pi,psi,sigm,snph, &
           start_dt=1.e15_iwp,theta,time,tol,tot_load,two=2.0_iwp,v,zero=0.0_iwp
CHARACTER(LEN=15)::argv;element='quadrilateral'; LOGICAL::converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::nf(:,:),g(:,),num(:,),g_num(:,:),g_g(:,:),etype(:,),  &
                      kdiag(:,),no(:,)

```

```

REAL(iwp,ALLOCATABLE::al(:),ans(:),bee(:,:,1),bdylds(:),bload(:),c(:,:,1), &
      coord(:,:,1),dee(:,:,1),der(:,:,1),derf(:,:,1),deriv(:,:,1),devp(:), &
      disp(:),eld(:),eload(:),eps(:),erate(:),evp(:),evpt(:,:,1),flow(:,:,1), &
      funf(:),g_coord(:,:,1),jac(:,:,1),kay(:,:,1),ke(:,:,1),km(:,:,1),kc(:,:,1),kv(:,:,1), &
      lf(:,:,1),loads(:),m1(:,:,1),m2(:,:,1),m3(:,:,1),newdis(:),oldis(:),phi0(:), &
      phi1(:),points(:,:,1),prop(:,:,1),sigma(:),store_kc(:,:,1),stress(:), &
      tensor(:,:,:),val(:,:,1),vol(:),volf(:,:,1),weights(:),x_coords(:), &
      y_coords(:))

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)
nx=nxe;nye=np_types; CALL mesh_size(element,nod,nel,nn,nxe,nye)
ALLOCATE(dee(nst,nst),points(nip,ndim),coord(nod,ndim),derivf(ndim,ndof), &
      jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod), &
      derf(ndim,ndof),funf(ndof),bee(ndof,ndof),km(ndof,ndof),eld(ndof), &
      sigma(ndof),kc(ndof,ndof),g_g(ntot,nel),ke(ntot,ntot),c(ndof,ndof), &
      x_coords(nxe+1),phi0(ndof),y_coords(nye+1),vol(ndof),nf(ndof,nn), &
      volf(ndof,ndof),g_coord(ndim,nn),g_num(nod,nel),num(nod),weights(nip), &
      phi1(ndof),store_kc(ndof,ndof,nel),tensor(nst+1,nip,nel),eps(nst), &
      evp(nst),evpt(nst,nip,nel),bload(ndof),eload(ndof),erate(nst),g(ntot), &
      devp(nst),m1(nst,nst),m2(nst,nst),m3(nst,nst),flow(nst,nst), &
      stress(nst),etype(nel),prop(np_props,np_types))
READ(10,*)
prop,cons; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtime,nstep,theta,npri,nres
nf=1; READ(10,*)nr,(k,nf(:,k)),i=1,nr; CALL formmf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),loads(0:neq),ans(0:neq),bdylds(0:neq),disp(0:neq), &
      newdis(0:neq),oldis(0:neq))
READ(10,*)loaded_nodes; ALLOCATE(no(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(no(i),val(i,:)),i=1,loaded_nodes)
READ(10,*)tol,limit,nlfp; ALLOCATE(lf(2,nlfp)); READ(10,*)lf
nls=FLOOR(lf(1,nlfp)/dtime); IF(nstep>nls)nstep=nls; ALLOCATE(al(nstep))
CALL load_function(lf,dtime,al); kdiag=0
!-----loop the elements to find global arrays sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'x')
    g(1:15:2)=nf(1,num(:)); g(2:16:2)=nf(2,num(:))
    g(17:20)=nf(3,num(1:7:2)); g_g(:,iel)=g
    g_num(:,iel)=num; g_coord(:,num)=TRANSPOSE(coord); CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO; ALLOCATE(kv(kdiag(neq)))
WRITE(11,'(2(A,I5))')
      " There are",neq," equations and the skyline storage is",kdiag(neq)
loads=zero; disp=zero; tensor=zero; kv=zero
CALL sample(element,points,weights)
!-----global matrix assembly-----
elements_2: DO iel=1,nels
    kay=zero; DO i=1,ndim; kay(i,i)=prop(i,etype(iel)); END DO
    CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    km=zero; c=zero; kc=zero
    gauss_points_1: DO i=1,nip
!-----elastic solid contribution-----
        CALL shape_der(der,points,i); jac=MATMUL(der,coord)
        det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
        tensor(1:2,i,iel)=cons; tensor(4,i,iel)=cons; tensor(5,i,iel)=zero
        CALL beemat(bee,deriv); vol(:)=bee(1,:)+bee(2,:)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!-----fluid contribution-----
    !-----fluid contribution-----

```

```

CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
derivf=MATMUL(jac,derf)
kc=kc+MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtim
DO k=1,nodf; volf(:,k)=vol(:)*funf(k); END DO; c=c+volf*det*weights(i)
END DO gauss_points_1; store_kc(:,:,iel)=kc
CALL formke(km,kc,c,ke,theta); CALL fsparv(kv,ke,g,kdiag)
END DO elements_2; CALL sparin_gauss(kv,kdiag) !---factorise equations---
pi=ACOS(-one); dt=start_dt
DO i=1,np_types
  phi=prop(5,i); snph=SIN(phi*pi/d180); e=prop(3,i); v=prop(4,i)
  ddt=d4*(one+v)*(one-two*v)/(e*(one-two*v+snph**2)); IF(ddt<dt)dt=ddt
END DO
!-----time stepping loop-----
oldis=zero; time=zero
WRITE(11,'(/A,I5)')" Results at node",nres
WRITE(11,'(A)')                                     &
"      time      load      x-disp      y-disp      porepressure iters"
WRITE(11,'(5E12.4)')0.0,0.0,0.0,0.0,0.0
time_steps: DO j=1,nstep; time=time+dtim; tot_load=SUM(al(1:j))
ans=zero; bdylds=zero; evpt=zero; newdis=zero
elements_3: DO iel=1,nels
  g=g_g(:,iel); kc=store_kc(:,:,iel); phi0=disps(g(ndof+1:)) ! gather
  phil=MATMUL(kc,phi0); ans(g(ndof+1:))=ans(g(ndof+1:))+phil ! scatter
END DO elements_3; ans(0)=zero; iters=0
!-----apply loading increment-----
DO i=1,loaded_nodes; ans(nf(1:2,no(i)))=val(i,:)*al(j); END DO
!-----plastic iteration loop-----
its: DO
  iters=iters+1; loads=ans+bdylds; CALL spabac_gauss(kv,loads,kdiag)
  WRITE(*,'(A,I6,A,I4)')" time step",j," iteration",iters
!-----check plastic convergence-----
newdis=loads; newdis(nf(3,:))=zero
CALL checon(newdis,oldis,tol,converged); IF(iters==1)converged=.FALSE.
IF(converged.OR.iters==limit)bdylds=zero
!-----go round the Gauss Points -----
elements_4: DO iel=1,nels; num=g_num(:,iel)
  coord=TRANSPOSE(g_coord(:,num)); phi=prop(5,etype(iel))
  coh=prop(6,etype(iel)); psi=prop(7,etype(iel))
  g=g_g(:,iel); eld=loads(g(1:ndof)); bload=zero
  CALL deemat(dee,prop(3,etype(iel)),prop(4,etype(iel)))
  gauss_points_2: DO i=1,nip; CALL shape_der(der,points,i)
    jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
    deriv=MATMUL(jac,der); CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
    eps=eps-evpt(:,i,iel); sigma=MATMUL(dee,eps)
    stress=sigma+tensor(1:4,i,iel)
    CALL invar(stress,sigm,dsbar,lode_theta)
  !-----check whether yield is violated-----
  CALL mocouf(phi,coh,sigm,dsbar,lode_theta,f)
  IF(converged.OR.iters==limit)THEN; devp=stress; ELSE
    IF(f>=zero)THEN
      CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
      CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
      erate=MATMUL(flow,stress); evp=erate*dt
      evpt(:,i,iel)=evpt(:,i,iel)+evp; devp=MATMUL(dee,evp)
    END IF
  END IF
  IF(f>=zero)THEN; eload=MATMUL(TRANSPOSE(bee),devp)
    bload=bload+eload*det*weights(i)
  END IF
END DO

```

```

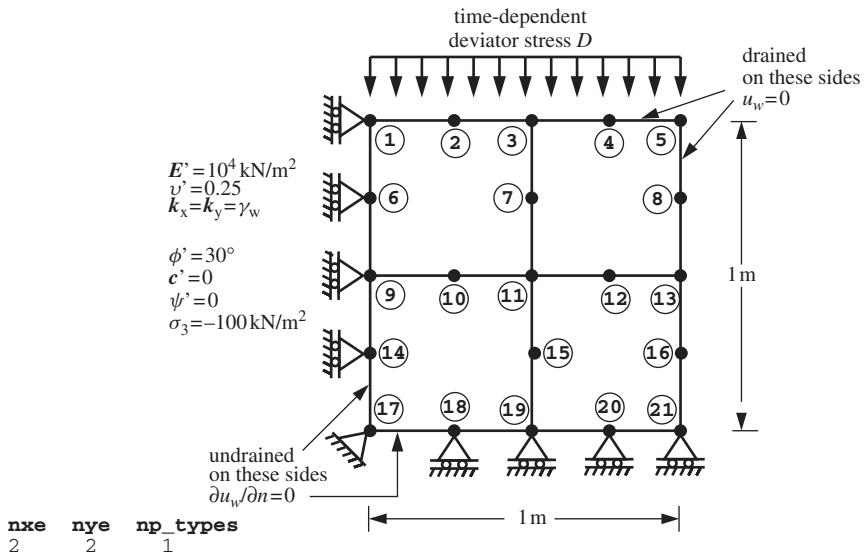
        END IF
        IF(converged.OR.iters==limit)THEN
!-----update the Gauss Point stresses and porepressures-----
        tensor(1:4,i,iel)=stress; dpore=zero
        CALL shape_fun(funf,points,i)
        DO k=1,nodf; dpore=dpore+funf(k)*loads(g(k+ndof)); END DO
        tensor(5,i,iel)=tensor(5,i,iel)+dpore
        END IF
    END DO gauss_points_2
!-----compute the total bodyloads vector -----
    bdylds(g(1:ndof))=bdylds(g(1:ndof))+bload
    END DO elements_4; bdylds(0)=zero; IF(converged.OR.iters==limit)EXIT
    END DO its; disp=disp+loads
    IF(j/npri*npri==j.OR.iters==limit)WRITE(11,'(5E12.4,I5)')      &
        time,tot_load,disp(nf(:,nres)),iters
    IF(iters==limit)EXIT
END DO time_steps
CALL dismsh(loads,nf,0.05_iwp,g_coord,g_num,argv,nlen,13)
CALL vecmsh(loads,nf,0.05_iwp,0.1_iwp,g_coord,g_num,argv,nlen,14)
STOP
END PROGRAM p96

```

There are 32 equations

Results at node 21						
time	load	x-disp	y-disp	porepressure	cg	iters
0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00		
0.1000E+00	0.2000E+00	0.0000E+00	0.0000E+00	-0.1994E+00		17
0.2000E+00	0.4000E+00	0.0000E+00	0.0000E+00	-0.3743E+00		17
0.3000E+00	0.6000E+00	0.0000E+00	0.0000E+00	-0.5125E+00		18
0.4000E+00	0.8000E+00	0.0000E+00	0.0000E+00	-0.6203E+00		19
0.5000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.7043E+00		17
0.6000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.5703E+00		14
0.7000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.4463E+00		13
0.8000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.3478E+00		15
0.9000E+00	0.1000E+01	0.0000E+00	0.0000E+00	-0.2709E+00		16
0.1000E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2110E+00		10
0.1100E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1643E+00		11
0.1200E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1280E+00		10
0.1300E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.9968E-01		9
0.1400E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.7764E-01		11
0.1500E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.6047E-01		12
0.1600E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.4710E-01		14
0.1700E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.3668E-01		11
0.1800E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2857E-01		10
0.1900E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2225E-01		13
0.2000E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1733E-01		5
0.2100E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1350E-01		10
0.2200E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1051E-01		14
0.2300E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.8187E-02		6
0.2400E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.6377E-02		12
0.2500E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.4966E-02		16
0.2600E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.3868E-02		10
0.2700E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.3013E-02		12
0.2800E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.2346E-02		14
0.2900E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1827E-02		14
0.3000E+01	0.1000E+01	0.0000E+00	0.0000E+00	-0.1423E-02		7

Figure 9.17 Results from Program 9.5 example



```

prop(k_x/gamma_w,k_y/gamma_w,e,v,phi,c,psi)
1.0e-6 1.0e-6 1.0e4 0.25 30.0 0.0 0.0

etype(not needed)

cons
-100.0

x_coords, y_coords
0.0 0.5 1.0
0.0 -0.5 -1.0

dtime nstep theta npri nres
0.5 200 0.5 1 1

nr, (k,nf(:,k),i=1,nr)
20
1 0 1 0 2 1 1 0 3 1 1 0 4 1 1 0 5 1 1 0
6 0 1 0 7 1 1 0 8 1 1 0 9 0 1 1 10 1 1 0
12 1 1 0 13 1 1 0 14 0 1 0 15 1 1 0 16 1 1 0
17 0 0 1 18 1 0 0 19 1 0 1 20 1 0 0 21 1 0 0

loaded_nodes, (no(i),val(i,:),i=1,loaded_nodes)
5
1 0.0 -0.08333 2 0.0 -0.33333 3 0.0 -0.16667
4 0.0 -0.33333 5 0.0 -0.08333

tol limit
0.001 250

nlfpp, (lf(:,i),i=1,nlfpp)
2
0.0 0.0 10.0 150.0

```

**Figure 9.18** Mesh and data for Program 9.6 example

The final program in this chapter is a non-linear version of Program 9.4 in which the material is modelled as an elastic–plastic material with a Mohr–Coulomb strength criterion. The program is an amalgamation of Programs 9.4 and 6.4 from Chapter 6, which used the viscoplastic strain method for redistributing excess internal stresses (e.g., Griffiths, 1994).

The three-dimensional array `tensor` is used to store the element integrating point stresses, with the four effective stress components coming first, followed by the pore water stress as the fifth ‘component’.

The illustrative problem shown in Figure 9.18 involves compression of a plane strain block of saturated elastic–plastic cohesionless soil by a time-dependent ‘deviator’ stress  $D$ , which is the difference between the vertical and (constant) horizontal stresses applied to the block.

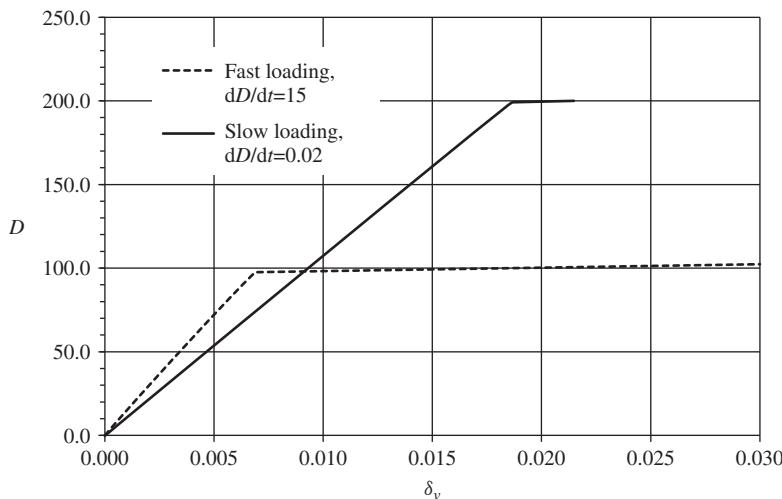
The data follows a similar course to that followed for Program 9.4. The number of properties has expanded to seven (`nprops=7`) with the addition of the friction angle  $\phi'$ , the cohesion  $c'$  and the dilation angle  $\psi$ . The ‘permeability’ property  $k/\gamma_w$  of the soil in this example is isotropic and set to  $1 \times 10^{-6}$ . The current example models a cohesionless soil with  $\phi' = 30^\circ$  and no dilation. The soil is initially consolidated to an isotropic compressive stress ( $\sigma_3$ ) of  $-100$  kN/m $^2$  read as `cons`. The coordinate data for `x_coords` and `y_coords` is followed by the familiar time-stepping and output control parameters. The current example calls for `nstep=200` calculation time steps of `dtime=0.5` s with `theta=0.5` (Crank–Nicolson). Output is required every time step (`npri=1`) at node `nres=1`. The `nr` data indicate 20 restrained nodes, which include rollers on the left and bottom boundaries, drainage conditions on the top and right boundaries, and removal of the third freedom at all mid-side nodes, as described in Program 9.4. The next data provide the load weightings corresponding to a unit pressure (Appendix A), to be applied to the five nodes at the top of the mesh. The next line reads the tolerance and iteration ceiling (`tol` and `limit`) for plastic iterations, as was first used in Program 4.5 and again extensively in Chapter 6. The final data define the load–time function to be applied at the top of the mesh. In this example the deviator stress is to increase linearly with time at a ‘fast’ rate given by  $dD/dt = 15$  kPa/s, so just two load function coordinates are required.

The results from this analysis are listed as Figure 9.19 and plotted in Figure 9.20, together with the results of a second analysis in which a ‘slow’ loading rate of

There are 36 equations and the skyline storage is 466

Results at node 1						
time	load	x-disp	y-disp	porepressure	iters	
0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00	0.0000E+00		
0.5000E+00	0.7500E+01	0.0000E+00	-0.5025E-03	0.0000E+00		2
0.1000E+01	0.1500E+02	0.0000E+00	-0.1009E-02	0.0000E+00		2
0.1500E+01	0.2250E+02	0.0000E+00	-0.1520E-02	0.0000E+00		2
0.2000E+01	0.3000E+02	0.0000E+00	-0.2035E-02	0.0000E+00		2
0.2500E+01	0.3750E+02	0.0000E+00	-0.2555E-02	0.0000E+00		2
0.3000E+01	0.4500E+02	0.0000E+00	-0.3078E-02	0.0000E+00		2
0.3500E+01	0.5250E+02	0.0000E+00	-0.3606E-02	0.0000E+00		2
0.4000E+01	0.6000E+02	0.0000E+00	-0.4138E-02	0.0000E+00		2
0.4500E+01	0.6750E+02	0.0000E+00	-0.4674E-02	0.0000E+00		2
0.5000E+01	0.7500E+02	0.0000E+00	-0.5213E-02	0.0000E+00		2
0.5500E+01	0.8250E+02	0.0000E+00	-0.5757E-02	0.0000E+00		2
0.6000E+01	0.9000E+02	0.0000E+00	-0.6304E-02	0.0000E+00		2
0.6500E+01	0.9750E+02	0.0000E+00	-0.6886E-02	0.0000E+00		9
0.7000E+01	0.1050E+03	0.0000E+00	-0.4326E-01	0.0000E+00		250

**Figure 9.19** Results from Program 9.6 example with  $dD/dt = 15$  kPa/s



**Figure 9.20** Deviator stress vs. axial displacement for different loading rates

$dD/dt = 0.02$  kPa/s was applied. The deviator stress at failure  $D_f$  was computed to be about 100 kPa for the ‘fast’ loading rate and 200 kPa for the ‘slow’ loading rate.

For plane strain compression of a non-dilative saturated soil,

$$D_f = \frac{\sigma_3(K_p - 1)(2\beta_{ps} + 1)}{(K_p + 1)\beta_{ps} + 1} \quad (9.7)$$

in which  $\beta_{ps} \rightarrow \infty$  and  $\beta_{ps} = 0$  give undrained and drained limiting conditions, respectively (Griffiths, 1985).

In this example,  $K_p = \tan^2(45^\circ + \phi'/2) = 3$ , so for a consolidating stress of  $\sigma_3 = 100$  kPa, (9.7) with  $\beta_{ps} \rightarrow \infty$  gives  $D_f \approx 100$  kPa, indicating that ‘fast’ loading is reproducing essentially undrained conditions. The ‘slow’ loading result of  $D_f \approx 200$  kPa corresponds to a drained solution, also given by (9.7) with  $\beta_{ps} = 0$ .

## 9.2 Glossary of Variable Names

### Scalar integers:

cg_iters	pcg or BiCGStab iteration counter
cg_limit	pcg or BiCGStab iteration ceiling
cg_tot	total number of BiCGStab iterations
ell	BiCGStab parameter; taken as 4 in this example
fixed Freedoms	number of fixed freedoms
i, iel	simple counters
iters	iteration counter
iwp	SELECTED_REAL_KIND(15)
j, k	simple counters
limit	iteration ceiling
loaded_nodes	number of loaded nodes
nband	full bandwidth of non-symmetric matrix

---

ndim	number of dimensions
ndof	number of displacement degrees of freedom per element
nels	number of elements
neq	number of degrees of freedom in the mesh
nlen	maximum number of characters in data file basename
nlfp	number of load function points
nls	maximum number of load steps
nip	number of integrating points
nn	number of nodes
nod	number of nodes per solid element
nodef	number of nodes per fluid element
nodof	number of degrees of freedom per node
npri	output printed every npri time steps
nprops	number of material properties
np_types	number of different property types
nr	number of restrained nodes
nres	node number at which time history is to be printed
nst	number of stress/strain terms
nstep	number of time steps required
ntime	time step number at which spatial distribution is to be printed
ntot	total number of degrees of freedom per element
nxe, nye	number of columns and rows of elements

**Scalar reals:**

at	holds area beneath isochrone by trapezoid rule at time $t$
a0	holds area beneath isochrone by trapezoid rule based on current load
alpha, beta	local variables
cg_tol	BiCGStab convergence tolerance
coh	soil cohesion
cons	consolidating stress ( $\sigma_3$ )
ddt	used to find the critical time step
det	determinant of the Jacobian matrix
dpore	holds the accumulated pore pressure
dq1	plastic potential derivative, $\partial Q / \partial \sigma_m$
dq2	plastic potential derivative, $\partial Q / \partial J_2$
dq3	plastic potential derivative, $\partial Q / \partial J_3$
dsbar	invariant, $\bar{\sigma}$
dt	critical viscoplastic time step
dtim	calculation time step
d4, d180	set to 4.0 and 180.0
e	Young's modulus
error	measure of error in BiCGStab
f	yield function
gama	local variable
hme	used to calculate ultimate consolidation settlement
kappa	BiCGStab parameter; taken as zero in this example

lode_theta	Lode angle, $\theta$
norm_r	residual norm
omega	local variable
one	set to 1.0
penalty	set to $1 \times 10^{20}$
phi	friction angle (degrees)
pi	set to $\pi$
psi	dilation angle (degrees)
pt5	set to 0.5
rho	fluid density
rho1	local variable
r0_norm	initial residual norm
settl	ultimate consolidation settlement
sigm	mean stress, $\sigma_m$
snph	sine of phi
start_dt	starting value of $\text{dt}$
theta	time-integration weighting parameter
time	holds time elapsed $t$
tol	convergence tolerance
two	set to 2.0
uav	average degree of consolidation based on excess pore pressure dissipation
uavs	average degree of consolidation based on settlement
ubar	average $x$ -velocity
up	holds dot product $\{\mathbf{R}\}_k^T \{\mathbf{R}\}_k$ from (3.22)
v	Poisson's ratio
x0	initialisation value
vbar	average $y$ -velocity
visc	molecular viscosity
zero	set to 0.0

**Scalar characters:**

argv	holds data file basename
element	element type

**Scalar logicals:**

converged	set to .TRUE. if solution converged
cg_converged	set to .TRUE. if BiCGStab has converged

**Dynamic integer arrays:**

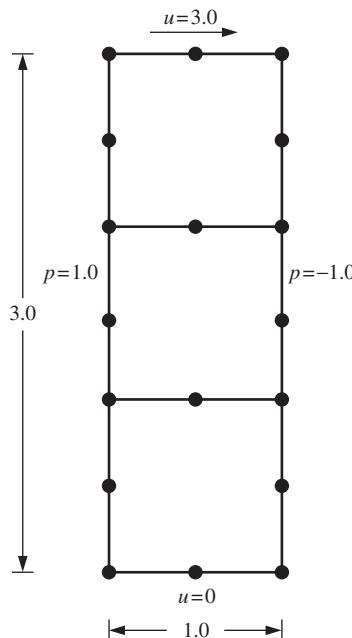
etype	element property types
g	element 'steering' vector
g_g	steering vector for all elements
g_num	node numbers for all elements
kdiag	diagonal term locations
nf	nodal freedoms
no	freedoms to be fixed
node	nodes with fixed displacement

num	element node numbers
sense	sense of freedom to be fixed
<b>Dynamic real arrays:</b>	
al	load steps at resolution of calculation time step
ans	rhs ‘load’ increment vector
b	right-hand-side vector
bdflds	self-equilibrating global body loads
bee	strain–displacement matrix
bload	self-equilibrating element body loads
c	coupling matrix
coord	solid element nodal coordinates
coordf	fluid element nodal coordinates
c11,c12,c21,c23,c32	element submatrix in (2.115)
d	vector used in (3.22)
dee	stress–strain matrix
der	solid shape function derivatives wrt local coordinates
derf	fluid shape function derivatives wrt local coordinates
deriv	solid shape function derivatives wrt global coordinates
derivf	fluid shape function derivatives wrt global coordinates
devp	product $[\mathbf{D}^e]\{\Delta\epsilon^{vp}\}$
diag	diagonal of left-hand-side matrix
diag_precon	diagonal preconditioner vector
disps	global displacements and pore pressures
eld	element nodal displacements
ell	element lengths
eload	integrating point contribution to bload
eps	strain terms
erate	viscoplastic strain rate, $\{\dot{\epsilon}^{vp}\}$
evp	viscoplastic strain rate increment, $\{\delta\epsilon^{vp}\}$
evpt	holds running total of viscoplastic strains, $\{\Delta\epsilon^{vp}\}$
flow	holds $\{\partial Q/\partial\sigma\}$
fun	solid shape functions
funf	fluid shape functions
gamma	small local array
gc	integrating point coordinates
gg	small local array
g_coord	nodal coordinates for all elements
jac	Jacobian matrix
kay	property matrix
kc	element conductivity matrix
kd	right-hand-side element matrix from ‘Biot’ analysis
ke	element ‘stiffness’ matrix
km	element stiffness matrix
kc	element conductivity matrix
kv	global stiffness matrix
lf	input load/time function
loads	nodal velocities and pressures

m1	holds $\partial\sigma_m/\partial\sigma$
m2	holds $\partial J_2/\partial\sigma$
m3	holds $\partial J_3/\partial\sigma$
nd1	product $[fun]^T[deriv(1,:)]$
nd2	product $[fun]^T[deriv(2,:)]$
ndf1	product $[fun]^T[derivf(1,:)]$
ndf2	product $[fun]^T[derivf(2,:)]$
nfd1	product $[funf]^T[deriv(1,:)]$
nfd2	product $[funf]^T[deriv(2,:)]$
newdis	‘new’ displacements and pore pressures
oldis	‘old’ displacements and pore pressures
oldlds	nodal velocities and pressures from previous iteration
p	‘descent’ vector used in equations (3.22)
pb	unsymmetric global band ‘stiffness’ matrix
phi0	used in element-by-element ‘gather’ algorithm
phi1	used in element-by-element ‘scatter’ algorithm
points	integrating point local coordinates
press	excess pore pressure values after ntime time steps
prop	element properties ( $E$ and $\nu$ for each element)
r	residual vector
rt	initial residual vector
s	small local vector
sigma, stress	stress terms
store	‘penalty’ degrees of freedom
storkd	stores augmented diagonal terms
storke	element matrix storage
store_kc	stores element kc matrices
tensor	holds running total of all integrating point stress terms
u, utemp	gather/scatter arrays
uvel	element nodal $x$ -velocity
val	nodal loads weighting factors
value	fixed vales of freedoms
vol	related to the volumetric strain
volf	used to compute coupling matrix
vvel	element nodal $y$ -velocity
weights	weighting coefficients
work	working space
x	‘old’ solution vector
xmul	gather/scatter array
xnew	‘new’ solution vector
x_coords	$x$ -coordinates of mesh layout
y, y1	gather/scatter arrays
y_coords	$y$ -coordinates of mesh layout

### 9.3 Exercises

- The mesh shown in Figure 9.21 is to be used to model 1D flow in the  $x$ -direction between two horizontal plates situated at  $y = 0.0$  and  $y = -3.0$ . Velocity boundary conditions are that the top plate is moved with a velocity of  $u = 3.0$  relative to the bottom plate, which is fixed at  $u = 0$ . Pressure boundary conditions are that the pressure on the left and right vertical boundaries is set to  $p = 1.0$  and  $p = -1.0$ , respectively, giving a pressure gradient of  $\partial p / \partial x = -2.0$ . Given that  $\mu = \rho = 1.0$  (visc and rho, respectively in programming terminology), use Program 9.1 to estimate the steady-state mid-plane velocity.



**Figure 9.21**

Answer:  $u = 3.75$

- Use Program 9.3 to reproduce the results shown in Figure 9.15.
- Use Program 9.4 to reproduce the Mandel–Cryer effect (see, e.g. Lambe and Whitman 1969, fig. 27.10), in which the excess pore pressures beneath the centre of a uniformly loaded flexible strip footing temporarily rise before they start to dissipate.
- Use a trial and error approach with the example accompanying Program 9.6 in this chapter to estimate the deviator stress loading rate that gives a failure load exactly half way between the drained and undrained solutions.

Answer:  $dD/dt \approx 2.17$  gives  $D_f \approx 150 \text{ kN/m}^2$

## References

- Biot MA 1941 General theory of three-dimensional consolidation. *J Appl Phys* **12**, 155–164.
- Griffiths DV 1985 The effect of pore fluid compressibility on failure loads in elasto-plastic soils. *Int J Numer Anal Methods Geomech* **9**, 253–259.
- Griffiths DV 1994 Coupled analyses in geomechanics. In *Visco-plastic Behavior of Geomaterials* (eds Cristescu ND and Gioda G). Springer-Verlag, Wien, pp. 245–317.
- Griffiths DV and Smith IM 2006 *Numerical Methods for Engineers*, 2nd edn. Chapman & Hall/CRC Press, Boca Raton, FL.
- Huang J and Griffiths DV 2010 One-dimensional consolidation theories for layered soil and coupled and uncoupled solutions by the finite-element method. *Géotechnique* **60**(9), 709–713.
- Kidger DJ 1994 Visualisation of three-dimensinal processes in geomechanics computations. In *Proc 8th Int Conf Comp Meths and Advvs GeoMech* (eds Siriwardane H and Zaman M). A. A. Balkema, Rotterdam, pp. 453–457.
- Lambe TW and Whitman RV 1969 *Soil Mechanics*. John Wiley & Sons, Chichester.
- Schiffman RL 1960 *Field applications of soil consolidation, time-dependent loading and variable permeability*. Technical Report 248, Highway Research Board, Washington, DC.
- Smith IM and Hobbs R 1976 Biot analysis of consolidation beneath embankments. *Géotechnique* **26**, 149–171.

# 10

## Eigenvalue Problems

### 10.1 Introduction

The ability to solve eigenvalue problems is important in many aspects of finite element work. For example, the number of zero eigenvalues of an element ‘stiffness’ matrix (its rank deficiency) is an important guide to the suitability of that element as illustrated in Chapter 3. In that context, the problem to be solved is just

$$[\mathbf{K}]\{\mathbf{u}\} = \lambda\{\mathbf{u}\} \quad (10.1)$$

which is the eigenvalue problem in ‘standard form’ (3.89). More often, the eigenvalue equation will describe a physical situation such as free vibration of a solid or fluid. For example, (2.19) after assembly, for a freely vibrating elastic solid becomes

$$[\mathbf{K}_m]\{\mathbf{X}\} = \omega^2[\mathbf{M}_m]\{\mathbf{X}\} \quad (10.2)$$

which can readily be converted to ‘standard form’ by the procedure outlined in Section 3.9.1. In this case the global mass matrix  $[\mathbf{M}_m]$  may be ‘lumped’ or ‘consistent’ (Section 3.7.7).

The present chapter describes four programs for the determination of eigenvalues and eigenvectors of such elastic structures and solids. Different algorithms and storage strategies are employed in the various cases. Since elastic solids are considered, the programs can be viewed as extensions of the programs described in Chapters 4 and 5. The same terminology is used. Program 10.1 computes the natural frequencies and mode shapes of strings of beam elements, and Program 10.2 does the same for planar elastic solids using 4- or 8-node quadrilaterals. Both programs use Jacobi’s method. Programs 10.3 and 10.4 use the Lanczos and Arnoldi algorithms, respectively (Section 3.9.2) to calculate natural frequencies and mode shapes of elastic solids. The first of these two programs uses a global assembly strategy while the second adopts an ‘element-by-element’ approach.

## Program 10.1 Eigenvalue analysis of elastic beams using 2-node beam elements. Lumped mass

```

PROGRAM p101
!-----
! Program 10.1 Eigenvalue analysis of elastic beams using 2-node
!           beam elements. Lumped mass.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,idiag,iel,ifail,j,k,nband,ndof=4,nels,neq,nlen,nmodes,nn,
nod=2,nodof=2,nprops=2,np_types,nr
REAL(iwp)::d12=12.0_iwp,one=1.0_iwp,pt5=0.5_iwp,penalty=1.e20_iwp,
etol=1.0e-30_iwp,zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,),g(:,),g_g(:,:,),kdiag(:,),nf(:,:,),num(:)
REAL(iwp),ALLOCATABLE::diag(:,),ell(:,),kh(:,),km(:,:,),ku(:,:,),kv(:,),
mm(:,:),prop(:,:,),rrmass(:,:,),udiag(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,np_types; nn=nels+1
ALLOCATE(nf(ndof,nn),km(ndof,ndof),num(nod),g(ndof),mm(ndof,ndof),
ell(nels),etype(nels),g_g(ndof,nels),prop(nprops,np_types))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype; READ(10,*)ell
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(diag(0:neq),udiag(0:neq),kdiag(neq),rrmass(0:neq))
!-----loop the elements to find global array sizes-----
nband=0; kdiag=0
elements_1: DO iel=1,nels
  num=(/iel,iel+1/); CALL num_to_g(num,nf,g); g_g(:,iel)=g
  IF(nband<bandwidth(g))nband=bandwidth(g); CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
WRITE(11,'(A,I5,A,/,A,I5,/,A,I5)')" There are",neq," equations",
" The half-bandwidth (including diagonal) is",nband+1,
" The skyline storage is",kdiag(neq)
!-----global stiffness and mass matrix assembly-----
ALLOCATE(ku(neq,nband+1),kv(kdiag(neq)),kh(kdiag(neq)))
diag=zero; ku=zero
elements_2: DO iel=1,nels
  g=g_g(:,iel); mm=zeros; mm(1,1)=pt5*prop(2,etype(iel))*ell(iel)
  mm(3,3)=mm(1,1); mm(2,2)=mm(1,1)*ell(iel)**2/d12; mm(4,4)=mm(2,2)
  CALL formlump(diag,mm,g); CALL beam_km(km,prop(1,etype(iel)),ell(iel))
  CALL formku(ku,km,g)
END DO elements_2
!-----reduce to standard eigenvalue problem-----
rrmass(1:)=one/SQRT(diag(1:))
DO i=1,neq; IF(i<=neq-nband)THEN; k=nband+1; ELSE; k=neq-i+1; END IF
  DO j=1,k; ku(i,j)=ku(i,j)*rrmass(i)*rrmass(i+j-1); END DO
END DO
!-----convert to skyline form-----
kh(1)=ku(1,1); k=1
DO i=2,neq; idiag=kdiag(i)-kdiag(i-1)
  DO j=1,idiag; k=k+1; kh(k)=ku(i+j-idiag,1-j+idiag); END DO
END DO

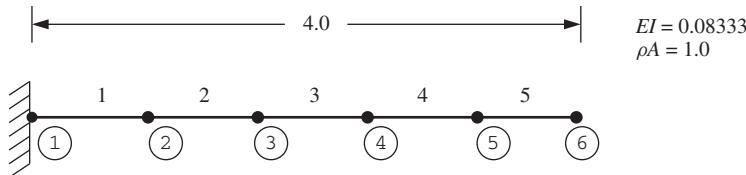
```

```

!-----extract the eigenvalues-----
CALL bandred(ku,diag,udiag); ifail=1; CALL bisect(diag,udiag,etol,ifail)
WRITE(11,'(/A)')" The eigenvalues are:"; WRITE(11,'(6E12.4)')diag(1:)
!-----extract the eigenvectors-----
READ(10,* )nmodes
DO i=1,nmodes
  kv=kh;kv(kdiag)=kv(kdiag)-diag(i); kv(1)=kv(1)+penalty
  udiag=zero; udiag(1)=kv(1)
  CALL sparin_gauss(kv,kdiag); CALL spabac_gauss(kv,udiag,kdiag)
  udiag=rmmass*udiag; WRITE(11,'(A,I3,A)')" Eigenvector number",i," is:"
  WRITE(11,'(6E12.4)')udiag(1:)/MAXVAL(ABS(udiag(1:)))
END DO
STOP
END PROGRAM p101

```

This program illustrates a natural frequency analysis of a typical ‘string’ of beam elements, and can be thought of as an extension to Program 4.3. The natural frequencies of the simple cantilever shown in Figure 10.1 are to be found. The first line of data gives the number of elements nels, which equals 5 in this case. The next line gives the number of properties np\_types which for a uniform beam equals 1. The two properties are then read in as the flexural stiffness  $EI$  read as 0.08333 and the mass per unit length  $\rho A$  read as 1.0. With only one property type, the etype data is not needed. The next line reads the lengths of the elements, which in this case are all equal to 0.8. The nodal freedom data then follows by fixing the cantilever end to have no translation or rotation. The final line of data reads nmodes which represents the number of eigenvectors (or mode shapes) to be computed. In this example the first three are requested.



```

nels
5

np_types
1

prop(ei,rhoa)
0.08333 1.0

etype(not needed)

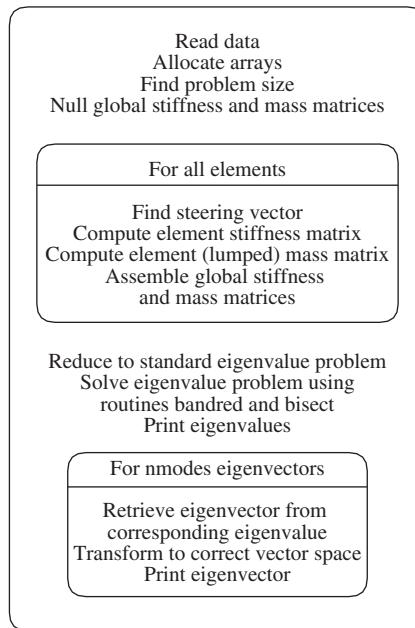
ell
0.8 0.8 0.8 0.8 0.8

nr,(k,nf(:,k),i=1,nr)
1
1 0 0

nmodes
3

```

**Figure 10.1** Mesh and data for Program 10.1 example



**Figure 10.2** Structure chart for Program 10.1

After the usual preliminary steps to establish the global array size, the elements are assembled into the global stiffness and mass matrices. In this case, the global stiffness  $[\mathbf{K}_m]$  is stored in `ku` as an upper band rectangle by library subroutine `formku` (see Figure 3.20), and the global lumped mass matrix  $[\mathbf{M}_m]$  is stored as a vector in `diag` by library subroutine `formlump`. The structure of the program is shown in Figure 10.2.

The diagonal element mass matrices  $[\mathbf{m}_m]$  are held in `mm` and use the following lumping (see, e.g., Cook *et al.*, 2002)

$$[\mathbf{m}_m] = \frac{\rho AL}{2} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & L^2/12 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & L^2/12 \end{bmatrix} \quad (10.3)$$

By factorising `diag` and altering the appropriate terms in `ku` (see Section 3.9.1), the symmetrical matrix for the standard eigenvalue problem is retrieved (still called `ku`). The eigenvalues of this band matrix ( $\omega^2$ ) are then calculated using Jacobi's method, which employs library subroutines `bandred` and `bisect`.

The first `nmodes` eigenvectors are then extracted by finding the relevant non-trivial solutions to the homogeneous equations and transforming them back to the correct vector space. In order to do this the global matrix is converted to skyline vector storage form (`kv`) and the eigenvectors solved for using subroutines `sparin_gauss` and `spabac_gauss`. In this example, the first three eigenvectors are computed and normalised to a vector length (Euclidean norm) of unity. The results are all shown in Figure 10.3.

```

There are    10 equations, the half-bandwidth is      3
and the skyline storage is    31

The eigenvalues are:
0.3823E-02  0.1278E+00  0.8511E+00  0.2765E+01  0.6323E+01  0.1147E+02
0.1748E+02  0.2324E+02  0.2756E+02  0.3225E+02
Eigenvector number 1 is:
0.6303E-01  0.1499E+00  0.2275E+00  0.2539E+00  0.4579E+00  0.3154E+00
0.7229E+00  0.3423E+00  0.1000E+01  0.3485E+00
Eigenvector number 2 is:
0.2307E+00  0.4535E+00  0.5431E+00  0.2347E+00  0.5094E+00 -0.3411E+00
0.2270E-01 -0.8308E+00 -0.7287E+00 -0.1000E+01
Eigenvector number 3 is:
0.2888E+00  0.4155E+00  0.3179E+00 -0.3976E+00 -0.1579E+00 -0.5696E+00
-0.2684E+00  0.3779E+00  0.3425E+00  0.1000E+01

```

**Figure 10.3** Results from Program 10.1 example

As a check, the computed results indicate a fundamental frequency  $\omega = \sqrt{0.0038} = 0.062$  which should be compared with the analytical result (e.g., Chopra, 1995) for a slender beam of

$$\omega = \frac{3.516}{L^2} \sqrt{\frac{EI}{\rho A}} = 0.063 \quad (10.4)$$

## Program 10.2 Eigenvalue analysis of an elastic solid in plane strain using 4- or 8-node rectangular quadrilaterals. Lumped mass. Mesh numbered in y-direction

```

PROGRAM p102
!-----
! Program 10.2 Eigenvalue analysis of an elastic solid in plane strain
! using 4- or 8-node rectangular quadrilaterals. Lumped mass.
! Mesh numbered in y-direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,idiag,iel,ifail,j,k,nband,ndim=2,ndof,nels,neq,nmodes,nn,nod, &
nodof=2,nlen,nprops=3,np_types,nr,nxe,nye
REAL(iwp)::area,etol=1.e-30_iwp,one=1.0_iwp,penalty=1.e20_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:),g_g (:,:),g_num(:,:),kdiag(:),nf(:,:,),&
num(:)
REAL(iwp),ALLOCATABLE::coord(:,:),diag(:),g_coord(:,:),kh(:),km(:,:,),&
ku(:,:,),kv(:,:,),mm(:,:,),prop(:,:,),rrmass(:,:,),udiag(:),x_coords(:),&
y_coords(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,nod,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye); ndof=nod*nodof
ALLOCATE(nf(ndof,nn),g_coord(ndim,nn),coord(nod,ndim),mm(ndof,ndof),&
g_num(nod,nels),num(nod),km(ndof,ndof),g(ndof),g_g(ndof,nels),&
prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype

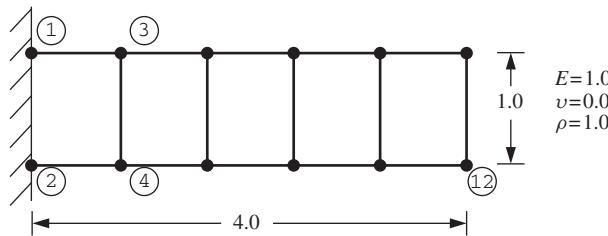
```

```

READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(diag(0:neq),udiag(0:neq),kdiag(neq),rrmass(0:neq))
!-----loop the elements to find global array sizes-----
nband=0; kdiag=0
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
    IF(nband<bandwidth(g))nband=bandwidth(g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
WRITE(11,'(A,I5,A,/,A,I5,/,A,I5)')" There are",neq," equations",      &
    " The half-bandwidth (including diagonal) is",nband+1,                  &
    " The skyline storage is",kdiag(neq)
!-----global stiffness and mass matrix assembly-----
ALLOCATE(ku(neq,nband+1),kv(kdiag(neq)),kh(kdiag(neq)))
diag=zero; ku=zero
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    CALL rect_km(km,coord,prop(1,etype(iel)),prop(2,etype(iel)))
    CALL formku(ku,km,g)
    area=(MAXVAL(coord(:,1))-MINVAL(coord(:,1)))*                      &
        (MAXVAL(coord(:,2))-MINVAL(coord(:,2)))
    CALL elmat(area,prop(3,etype(iel)),mm); CALL formlump(diag,mm,g)
END DO elements_2
!-----reduce to standard eigenvalue problem-----
rrmass(1:)=one/SQRT(diag(1:))
DO i=1,neq; IF(i<=neq-nband)THEN; k=nband+1; ELSE; k=neq-i+1; END IF
    DO j=1,k; ku(i,j)=ku(i,j)*rrmass(i)*rrmass(i+j-1); END DO
END DO
!-----convert to skyline form-----
kh(1)=ku(1,1); k=1
DO i=2,neq; iddiag=kdiag(i)-kdiag(i-1)
    DO j=1,iddiag; k=k+1; kh(k)=ku(i+j-iddiag,1-j+iddiag); END DO
END DO
!-----extract the eigenvalues-----
CALL bandred(ku,diag,udiag); ifail=1; CALL bisect(diag,udiag,etol,ifail)
WRITE(11,'(/A)')" The eigenvalues are:"; WRITE(11,'(6E12.4)')diag(1:)
!-----extract the eigenvectors-----
READ(10,*)nmodes
DO i=1,nmodes
    kv=kh;kv(kdiag)=kv(kdiag)-diag(i); kv(1)=kv(1)+penalty
    udiag=zero; udiag(1)=kv(1)
    CALL sparin_gauss(kv,kdiag); CALL spabac_gauss(kv,udiag,kdiag)
    udiag=rrmass*udiag; WRITE(11,'(A,I3,A)')" Eigenvector number",i," is:"
    WRITE(11,'(6E12.4)')udiag(1:)/MAXVAL(ABS(udiag(1:)))
    IF(i==1)CALL dismsh(udiag,nf,0.1_iwp,g_coord,g_num,argv,nlen,13)
END DO
STOP
END PROGRAM p102

```

This program is an extension of Program 5.1 for the analysis of elastic solids in plane strain. The element coordinates and steering information are produced by the geometry subroutine `geom_rect`, otherwise the structure of the program has much in common with Program 10.1 (see Figure 10.2).



```

nxe nye nod
5 1 4

np_types
1

prop(e,v,rho)
1.0 0.0 1.0

etype (not needed)

x_coords, y_coords
0.0 0.8 1.6 2.4 3.2 4.0
0.0 -1.0

nr,(k,nf(:,k),i=1,nr)
2
1 0 0 2 0 0

nmodes
5

```

**Figure 10.4** Mesh and data for first Program 10.2 example

The example problem shown in Figure 10.4 is nominally the same as the beam analysed in Figure 10.1, namely an elastic solid cantilever 4.0 units long in the  $x$ -direction with a flexural rigidity of 0.08333. The solid modelled by the five 4-node elements in Figure 10.4 is two-dimensional, however, so Poisson's ratio has been set to zero to remove the stiffening effect of plane strain. The mass density is set to unity.

Rather than performing the usual numerical integration loops, this program introduces the new subroutine `rect_km` which computes the stiffness matrix (`km`) of an elastic 4- or 8-node rectangular element in 'closed form' based on `nip=4`. Subroutine `elmat` forms the lumped mass matrix (`mm`) for a 4- or 8-node quadrilateral based on its area. For a 4-node element, the lumped mass matrix `mm` is readily formed with eight diagonal terms, in which one-quarter of the total mass of the element is lumped at each node in each direction. The element stiffness and mass matrices are assembled into their global counterparts `ku` and `diag` as discussed previously, and the remainder of the program is identical to Program 10.1.

The first line of data provides the number of elements in the  $x$ - and  $y$ -directions (`nxe` and `nye`) and the number of nodes per element (`nod`). Three properties are required in a problem such as this, namely, Young's modulus  $E$ , Poisson's ratio  $\nu$  and the mass density  $\rho$ . This is followed by the mesh coordinate data (`x_coords` and `y_coords`) and the boundary condition data which involves fully fixing the nodes at the built-in end of the cantilever. The final line of data as before reads `nmodes`, which represents

There are 20 equations, the half-bandwidth is 7  
and the skyline storage is 114

The eigenvalues are:

0.4595E-02	0.1053E+00	0.1529E+00	0.5169E+00	0.1226E+01	0.1288E+01
0.2058E+01	0.2352E+01	0.2415E+01	0.2710E+01	0.2719E+01	0.3114E+01
0.3125E+01	0.3188E+01	0.3381E+01	0.3665E+01	0.3960E+01	0.4211E+01
0.4962E+01	0.6097E+01				

Eigenvector number 1 is:

0.7195E-01	-0.6932E-01	-0.7195E-01	-0.6932E-01	0.1224E+00	-0.2362E+00
-0.1224E+00	-0.2362E+00	0.1526E+00	-0.4662E+00	-0.1526E+00	-0.4662E+00
0.1662E+00	-0.7285E+00	-0.1662E+00	-0.7285E+00	0.1695E+00	-0.1000E+01
-0.1695E+00	-0.1000E+01				

Eigenvector number 2 is:

0.2487E+00	-0.4240E+00	-0.2487E+00	-0.4240E+00	0.1022E+00	-0.8727E+00
-0.1022E+00	-0.8727E+00	-0.2584E+00	-0.7980E+00	0.2584E+00	-0.7980E+00
-0.5732E+00	-0.7556E-01	0.5732E+00	-0.7556E-01	-0.6870E+00	0.1000E+01
0.6870E+00	0.1000E+01				

Eigenvector number 3 is:

0.3090E+00	0.1555E-14	0.3090E+00	0.1572E-14	0.5878E+00	0.3059E-14
0.5878E+00	0.2986E-14	0.8090E+00	0.2674E-14	0.8090E+00	0.2728E-14
0.9511E+00	-0.5347E-16	0.9511E+00	0.0000E+00	0.1000E+01	-0.3174E-14
0.1000E+01	-0.3478E-14				

Eigenvector number 4 is:

0.2413E+00	-0.9151E+00	-0.2413E+00	-0.9151E+00	-0.4612E+00	-0.8557E+00
0.4612E+00	-0.8557E+00	-0.5141E+00	0.3742E+00	0.5141E+00	0.3742E+00
0.4120E+00	0.6579E+00	-0.4120E+00	0.6579E+00	0.1000E+01	-0.7050E+00
-0.1000E+01	-0.7050E+00				

Eigenvector number 5 is:

0.2575E+00	0.9693E+00	-0.2575E+00	0.9693E+00	0.8105E+00	-0.2304E+00
-0.8105E+00	-0.2304E+00	-0.3806E+00	-0.5583E+00	0.3806E+00	-0.5583E+00
-0.6787E-01	0.6923E+00	0.6787E-01	0.6923E+00	0.1000E+01	-0.2476E+00
-0.1000E+01	-0.2476E+00				

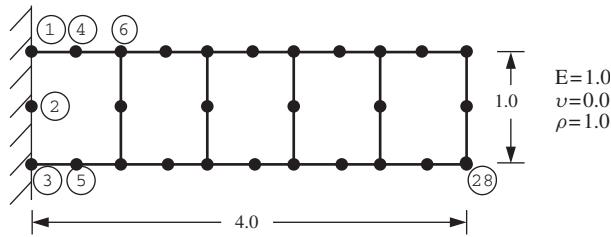
**Figure 10.5** Results from first Program 10.2 example

the number of eigenvectors (or mode shapes) to be computed. In this example, five eigenvectors are requested.

From the results shown in Figure 10.5, it can be seen that the fundamental frequency, printed as  $\omega = \sqrt{0.004595} = 0.068$ , is rather higher than the value of 0.062 calculated by Program 10.1 for a slender beam. Thus, the 2D solid, represented by 4-node elements with full integration, is a poor representation of a slender beam, at least in the flexural modes. The elements are too ‘stiff’. The longitudinal modes as indicated by the third eigenvalue and eigenvector are more accurately modelled by this element, however; the computed frequency, printed as  $\omega = \sqrt{0.1529} = 0.391$ , is in good agreement with the analytical solution of  $\omega = \pi/2L\sqrt{E/\rho} = 0.393$ .

A much better representation of flexural modes of ‘beams’ made up of solid elements is achieved by the use of 8-node quadrilaterals, so the second example uses this superior element to solve the same problem, however the process of mass lumping is not obvious in this case. For example, the summation of rows of the consistent matrix leads to negative values at the corners. It can be shown, however (e.g., Smith, 1977) that a reasonable approximation is to lump the mass to the mid-point and corner nodes in the ratio 4:1, thus 1/20 of the total mass is assigned to each corner node and 1/5 to each mid-side node in each direction. This weighting is assigned to the 8-node element by subroutine elmat.

The mesh and data shown in Figure 10.6 are virtually identical to those used for the analysis with 4-node elements in Figure 10.4. The only differences lie in the data for nod, which is now read as 8 and the boundary condition data at the built-in end of the cantilever, which now involves three fixed nodes.



```

nx e   ny e   nod
5      1      8

np_types
1

prop(e,v,rho)
1.0  0.0  1.0

etype (not needed)

x_coords, y_coords
0.0  0.8  1.6  2.4  3.2  4.0
0.0 -1.0

nr, (k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

nmodes
5

```

**Figure 10.6** Mesh and data for second Program 10.2 example

The output shown in Figure 10.7 indicates a fundamental frequency of  $\omega = \sqrt{0.003641} = 0.060$ , which is in closer agreement with the value of 0.062 produced by Program 10.1. The program outputs the fundamental mode shape to the graphics output file `*.dis` and this is shown in Figure 10.8.

### 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

```

PROGRAM p103
!-----
! Program 10.3: Eigenvalue analysis of an elastic solid in plane strain
!                 using 4-node rectangular quadrilaterals. Lanczos Method.
!                 Consistent mass. Mesh numbered in y-direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,iflag=-1,iters,jflag,k,lalpha,leig,lp=6,lx,1z,nband=0,      &
ndim=2,ndof,neig=0,nels,neq,nip=4,nlen,nmodes,nn,nod,nodof=2,nprops=3,      &
np_types,nr,nst=3,nxe,nye; REAL(iwp)::acc,det,el,er,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
!----- dynamic arrays-----

```

```

INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:,g_num(:,:)),jeig(:,:,nf(:,:,&
nu(:),num(:)

REAL(iwp),ALLOCATABLE::alfa(:,bee(:,beta(:,coord(:,:,dee(:,&
del(:,der(:),deriv(:),diag(:,ecm(:,eig(:,fun(:,g_coord(:,&
jac(:,kb(:,:),km(:,:),mb(:,:),mm(:,:),points(:,:),prop(:,ua(:,&
udiag(:,va(:,v_store(:,:),weights(:,w1(:,x(:,x_coords(:,y(:,:),&
y_coords(:,z(:,:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nx,e,ye,nod,np_types
CALL mesh_size(element,nod,nels,nn,nxe,ye); ndof=nod*nodof
ALLOCATE(nf(ndof,nn),points(nip,ndim),dee(nst,nst),g_coord(ndim,nn),&
coord(nod,ndim),fun(nod),jac(ndim,ndim),weights(nip),g_num(nod,nels),&
der(ndim,nod),deriv(ndim,nod),bee(nst,nod),num(nod),km(ndof,ndof),&
g(ndof),g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),&
prop(nprops,np_types),x_coords(nx+1),y_coords(ye+1),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
READ(10,*)nmodes,el,er,lalpha,leig,lx,lz,acc
ALLOCATE(eig(leig),x(lx),del(lx),nu(lx),jeig(2,leig),alfa(lalpha),&
beta(lalpha),z(lz,leig))
!----- loop the elements to find nband and set up global arrays -----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
    IF(nband<bandwidth(g))nband=bandwidth(g)
END DO elements_1
WRITE(11,'(A,I5,A,/,A,I5,,A,I5)')" There are",neq," equations",&
" The half-bandwidth (including diagonal) is",nband+1
ALLOCATE(kb(neq,nband+1),mb(neq,nband+1),ua(0:neq),va(0:neq),&
diag(0:neq),udiag(0:neq),w1(0:neq),y(0:neq,leig),v_store(0:neq,lalpha))
kb=zero; mb=zero; ua=zeros; va=zeros; eig=zeros; jeig=0; x=zeros; del=zeros
nu=0; alfa=zeros; beta=zeros; diag=zeros; udiag=zeros; w1=zeros; y=zeros; z=zeros
CALL sample(element,points,weights)
!----- element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    km=zero; mm=zero
    integrating_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); call beemat(bee,deriv)
        km=km+MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        call ecmat(ecm,fun,ndof,nodof)
        mm=mm+ecm*det*weights(i)*prop(3,etype(iel))
    END DO integrating_pts_1
    CALL formkb(kb,km,g); CALL formkb(mb,mm,g)
END DO elements_2; CALL cholinc(mb)
!-----find eigenvalues-----
DO iters=1,lalpha
    CALL lancz1(neq,el,er,acc,leig,lx,lalpha,lp,iflag,ua,va,eig,jeig,neig,x,&
    del,nu,alfa,beta,v_store)
    IF(iflag==0)EXIT
    IF(iflag>1)THEN; WRITE(11,'(A,I5)')&
    " Lancz1 is signalling failure, with iflag = ", iflag; STOP

```

```

    END IF
!--- iflag = 1 therefore form u + a * v ( candidate for ebe ) -----
    udiag=va; CALL chobk2(mb,udia); CALL banmul(kb,udia,diag)
    CALL chobk1(mb,diag); ua=ua+diag
END DO
!--- iflag = 0 therefore write out the spectrum -----
WRITE(11,'(A,I5,A/)')" It took",iters," iterations"
WRITE(11,'(3(A,E12.4))')" Eigenvalues in the range",el," and",er," are:"
WRITE(11,'(6E12.4)')eig(1:neig)
!----- calculate the eigenvectors -----
IF(neig>10)neig=10
    CALL lancz2(neq,lalfa,lp,eig,jeig,neig,alfa,beta,lz,jflag,y,w1,z,v_store)
!-----if jflag is zero calculate the eigenvectors -----
IF(jflag==0)THEN
    DO i=1,nmodes
        udiag(:,i)=y(:,i); CALL chobk2(mb,udia)
        WRITE(11,'(" Eigenvector number",I4," is:")')i
        WRITE(11,'(6E12.4)')udia(1:)/MAXVAL(ABS(udia(1:)))
    END DO
ELSE
! lancz2 fails
    WRITE(11,'(A,I5)')" Lancz2 is signalling failure with jflag = ",jflag
END IF
STOP
END PROGRAM p103

```

There are 50 equations, the half-bandwidth is 15  
and the skyline storage is 515

The eigenvalues are:

0.3641E-02	0.9363E-01	0.1532E+00	0.4932E+00	0.1255E+01	0.1270E+01
0.1524E+01	0.2390E+01	0.3006E+01	0.3321E+01	0.3803E+01	0.4927E+01
0.5010E+01	0.5648E+01	0.5844E+01	0.6508E+01	0.6935E+01	0.7054E+01
0.7097E+01	0.7420E+01	0.9235E+01	0.9569E+01	0.9828E+01	0.9873E+01
0.1050E+02	0.1056E+02	0.1079E+02	0.1082E+02	0.1101E+02	0.1119E+02
0.1124E+02	0.1152E+02	0.1172E+02	0.1178E+02	0.1388E+02	0.1582E+02
0.1634E+02	0.1793E+02	0.1915E+02	0.1998E+02	0.2072E+02	0.2263E+02
0.2639E+02	0.3313E+02	0.3879E+02	0.4823E+02	0.5123E+02	0.5894E+02
0.6294E+02	0.7198E+02				

Eigenvector number 1 is:

0.3907E-01	-0.2065E-01	-0.3907E-01	-0.2065E-01	0.7253E-01	-0.7030E-01
-0.2551E-13	-0.7029E-01	-0.7253E-01	-0.7030E-01	0.1004E+00	-0.1444E+00
-0.1004E+00	-0.1444E+00	0.1228E+00	-0.2385E+00	-0.3805E-13	-0.2384E+00
-0.1228E+00	-0.2385E+00	0.1401E+00	-0.3480E+00	-0.1401E+00	-0.3480E+00
0.1524E+00	-0.4691E+00	-0.3349E-13	-0.4690E+00	-0.1524E+00	-0.4691E+00
0.1606E+00	-0.5976E+00	-0.1606E+00	-0.5976E+00	0.1652E+00	-0.7307E+00
-0.3567E-13	-0.7306E+00	-0.1652E+00	-0.7307E+00	0.1672E+00	-0.8651E+00
-0.1672E+00	-0.8651E+00	0.1677E+00	-0.1000E+01	-0.3424E-13	-0.9998E+00
-0.1677E+00	-0.1000E+01				

.

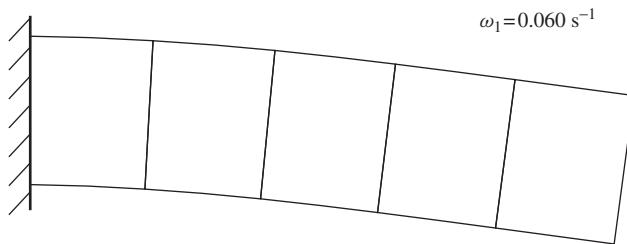
.

.

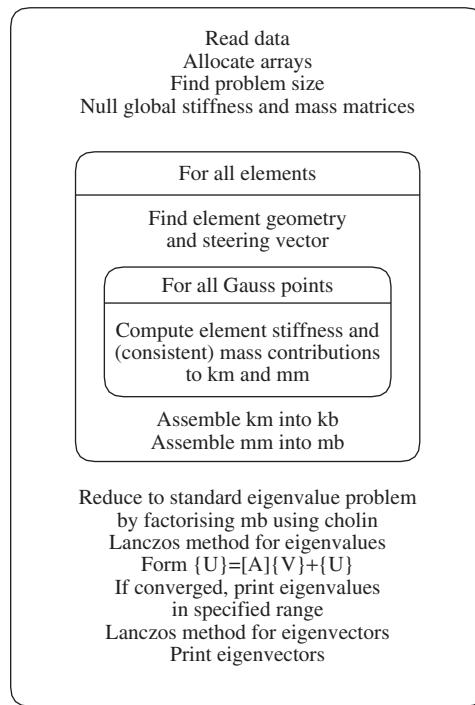
Eigenvector number 5 is:

0.1827E+00	0.6303E-02	0.1827E+00	-0.6303E-02	0.3434E+00	0.1255E-01
0.3188E+00	-0.1552E-12	0.3434E+00	-0.1255E-01	0.4397E+00	0.2189E-01
0.4397E+00	-0.2189E-01	0.4831E+00	-0.1363E-01	0.3530E+00	0.7292E-13
0.4831E+00	0.1363E-01	0.2961E+00	-0.5326E-01	0.2961E+00	0.5326E-01
0.7352E-01	0.1180E-01	0.1981E+00	0.8243E-13	0.7352E-01	-0.1180E-01
0.2596E-01	0.8228E-01	0.2596E-01	-0.8228E-01	-0.2472E-01	-0.3080E-01
-0.3033E+00	-0.1238E-12	-0.2472E-01	0.3080E-01	-0.5452E+00	-0.1962E+00
-0.5452E+00	0.1962E+00	-0.1000E+01	0.5836E+00	-0.8896E-01	0.1023E-12
-0.1000E+01	-0.5836E+00				

Figure 10.7 Results from second Program 10.2 example



**Figure 10.8** Fundamental mode shape from second Program 10.2 example



**Figure 10.9** Structure chart for Program 10.3

In Programs 10.1 and 10.2, Jacobi transformation was used to solve the eigenvalue problem. Although reliable and robust, this method is time-consuming for large problems. For such problems, iterative methods are more attractive, for example the Lanczos method (e.g., Parlett and Reid, 1981). The process is described in Chapter 3, Section 3.9.2, and is used in the present program, whose structure chart is given as Figure 10.9. It should be noted that for the first time in this chapter, consistent mass has been used. The global mass matrix is therefore symmetric and banded, and stored in array mb in the same way that the global stiffness matrix is stored in array kb. Assembly is achieved using the subroutine formkb (see Table 3.7 and Figure 3.20).

```

nxe  nye  nod
5      1      4

np_types
1

prop (e,v,rho)
1.0  0.0  1.0

etype (not needed)

x_coords, y_coords
0.0  0.8  1.6  2.4  3.2  4.0
0.0 -1.0

nr, (k,nf(:,k),i=1,nr)
2
1 0 0  2 0 0

nmodes
5

el      er    lalpha   leig   lx     lz     acc
0.0    5.0   500      20     80    500   1.0E-6

```

**Figure 10.10** Data for Program 10.3 example

The principal new vectors are  $\{\mathbf{U}\}$  and  $\{\mathbf{V}\}$  (called  $\mathbf{ua}$  and  $\mathbf{va}$  in the program), as described in the structure chart of Figure 10.9. Library subroutines `lancz1` and `lancz2`<sup>1</sup> are used (see also Appendix G) to retrieve the eigenvalues and eigenvectors, respectively (e.g., Smith and Heshmati, 1983; Smith, 1984).

The same 4-node problem considered previously is used to demonstrate this program with data as shown in Figure 10.10. Additional data read involves `el` and `er`, which define the range of the eigenvalue search and `lalpha`, `leig`, `lx`, `lz` and `acc`, which are parameters related to the Lanczos algorithm (see Glossary at the end of this chapter).

The output from the program is listed as Figure 10.11. It consists of the number of Lanczos iterations (22 in this case) and the computed eigenvalues in the range  $0.0 < \omega^2 < 5.0$ . The first five eigenvectors are also printed.

Since the consistent mass assumption was made, the eigenvalues differ somewhat from those computed by Program 10.2. For example, the first eigenvalue is computed as  $\omega = \sqrt{0.004931} = 0.070$  compared with 0.068 previously computed for a similar problem with lumped mass.

<sup>1</sup> These routines were developed from the Harwell Subroutine Library and readers should refer to HSL (2011) for further details and conditions of use.

```

There are 20 equations and the half-bandwidth is      7
It took 22 iterations

Eigenvalues in the range 0.0000E+00 and 0.5000E+01 are:
 0.4931E-02  0.1499E+00  0.1555E+00  0.9573E+00  0.1493E+01  0.3043E+01
 0.4688E+01
Eigenvector number 1 is:
-0.7275E-01  0.7082E-01  0.7275E-01  0.7082E-01 -0.1229E+00  0.2394E+00
 0.1229E+00  0.2394E+00 -0.1522E+00  0.4700E+00  0.1522E+00  0.4700E+00
-0.1648E+00  0.7311E+00  0.1648E+00  0.7311E+00 -0.1677E+00  0.1000E+01
 0.1677E+00  0.1000E+01
Eigenvector number 2 is:
-0.2156E+00  0.4086E+00  0.2156E+00  0.4086E+00 -0.6107E-01  0.7885E+00
 0.6107E-01  0.7885E+00  0.2617E+00  0.6519E+00 -0.2617E+00  0.6519E+00
 0.5123E+00 -0.5054E-01 -0.5123E+00 -0.5054E-01  0.5925E+00 -0.1000E+01
-0.5925E+00 -0.1000E+01
Eigenvector number 3 is:
 0.3090E+00 -0.2022E-06  0.3090E+00 -0.2015E-06  0.5878E+00 -0.3972E-06
 0.5878E+00 -0.3961E-06  0.8090E+00 -0.3516E-06  0.8090E+00 -0.3508E-06
 0.9511E+00 -0.3366E-07  0.9511E+00 -0.3273E-07  0.1000E+01  0.4045E-06
 0.1000E+01  0.4053E-06
Eigenvector number 4 is:
-0.1193E+00  0.9255E+00  0.1193E+00  0.9255E+00  0.4733E+00  0.5949E+00
-0.4733E+00  0.5949E+00  0.3215E+00 -0.6339E+00 -0.3215E+00 -0.6339E+00
-0.5413E+00 -0.5437E+00  0.5413E+00 -0.5437E+00 -0.1000E+01  0.9777E+00
 0.1000E+01  0.9777E+00
Eigenvector number 5 is:
-0.8090E+00 -0.4830E-07 -0.8090E+00 -0.5335E-07 -0.9511E+00 -0.4265E-07
-0.9511E+00 -0.4216E-07 -0.3090E+00  0.3175E-07 -0.3090E+00  0.3655E-07
 0.5878E+00  0.3762E-07  0.5878E+00  0.4084E-07  0.1000E+01 -0.7482E-07
 0.1000E+01 -0.7508E-07

```

**Figure 10.11** Results from Program 10.3 example

### 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

```

PROGRAM p104
!-----
! Program 10.4: Eigenvalue analysis of an elastic solid in plane strain
!                 using 4-node rectangular quadrilaterals.
!                 Arnoldi's Method. Lumped mass. Element by element.
!                 Mesh numbered in y-direction.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,ido,iel,ierr,info,iparam(11),ipntr(11),ishfts,iters,j,k,      &
logfil,lworkl,maxitr,model,msaitr,msapps,msaupd,msaup2,mseigt,mseupd,      &
msgsets,ndigit,nconv,ncv,ndim=2,nev,ndof,nels,neg,nip=4,nlen,nn,nod,      &
nodof=2,nprops=3,np_types,nr,nst=3,nxe,nye
REAL(iwp)::det,one=1.0_iwp,sigma,tol,zero=0.0_iwp
CHARACTER(len=15)::argv,bmat,element='quadrilateral',which; LOGICAL::rvec
!----- dynamic arrays-----
REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),d(:,:),dee(:,:),der(:,:),
  deriv(:,:),diag(:),ecm(:,:),fun(:),g_coord(:,:),jac(:,:),km(:,:),
  mm(:,:),pmul(:),points(:,:),prop(:,:),resid(:),udiag(:),utemp(:),
  v(:,:),vdiag(:),weights(:),workd(:),workl(:),x_coords(:),y_coords(:)
INTEGER,ALLOCATABLE::etype(:),g(:,g_g (:,:)),g_num(:,:),nf(:,:),num(:)
LOGICAL,ALLOCATABLE::select(:)

```

```

! include 'debug.h'
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,nod,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye);ndof=nod*nodof
ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),g_coord(ndim,nn) , &
coord(nod,ndim),fun(nod),jac(ndim,ndim), weights(nip),utemp(ndof), &
g_num(nod,nels),der(ndim,nod),deriv(ndim,nod),bee(nst,ndof),num(nod), &
km(ndof,ndof),g(ndof),g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof), &
pmul(ndof),x_coords(nxe+1),y_coords(nye+1),prop(nprops,np_types), &
etype(nels)); READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
READ(10,*)nev,ncv,bmat,which,tol,maxitr
lworkl=ncv*(ncv+8); ndigit=-3; logfil=6; msgsets=0; msaitr=0; msapps=0
msaupd=2; msaup2=0; mseigt=0; mseupd=0
! msaupd=1; msaup2=0; mseigt=0; mseupd=0
info=0; ido=0; ishfts=1; model=1; ipntr=0; iparam=0
iparam(1)=ishfts; iparam(3)=maxitr; iparam(7)=model
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g (num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=transpose(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
WRITE(11,'(A,I5,A/)')" There are",neq," equations"
ALLOCATE(v(neq,ncv),diag(neq),udiag(neq),vdiag(neq),workd(3*neq) , &
resid(neq),workl(lworkl),d(ncv,2),select(ncv))
diag=zero; v=zero; udiag=zero; vdiag=zero; workd=zero; resid=zero
CALL sample(element,points,weights)
!----- element stiffness integration and assembly-----
elements_2: DO iel=1,nels; num=g_num(:,iel)
    coord=transpose(g_coord(:,num )); g=g_g(:,iel); km=zero; mm=zero
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    integrating_pts_1: DO i=1,nip; CALL shape_fun(fun,points,i)
        CALL shape_der(der,points,i); jac=MATMUL(der,coord)
        det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        CALL ecmat(ecm,fun,ndof,nodof)
        mm=mm+ecm*det*weights(i)**prop(3,etype(iel))
    END DO integrating_pts_1
    DO i=1,ndof; if(g(i)/=0)diag(g(i))=diag(g(i))+sum(mm(i,:)); END DO
END DO elements_2
!-----find eigenvalues-----
iters=0; diag=one/SQRT(diag)      ! diag holds  $1^{**(-1/2)}$ 
DO
    iters=iters+1
    CALL dsaupd(ido,bmat,neq,which,nev,tol,resid,ncv,v,neq,iparam,ipntr, &
    workd,workl,lworkl,info)
    IF(ido/=-1.AND.ido/=1)EXIT
    udiag=zero; vdiag=workd(ipntr(1):ipntr(1)+neq-1)*diag
    elements_3: DO iel=1,nels; g=g_g(:,iel)
        DO i=1,ndof
            IF(g(i)==0)pmul(i)=zero; IF(g(i)/=0)pmul(i)=vdiag(g(i))
        END DO; utemp=matmul(km,pmul)
        DO i=1,ndof; IF(g(i)/=0)udiag(g(i))=udiag(g(i))+utemp(i); END DO
    END DO
END DO

```

```

END DO elements_3
  udiag=udiaig*diag; workd(ipntr(2):ipntr(2)+neq-1)=udiaig
END DO
WRITE(11,'(A,I5,A/)' )" Convergence after ",iters," iterations"
!-----Either we have convergence or there is an error-----
IF(info<0)THEN; WRITE(11,*)"Fatal error in ssaupd "
ELSE; rvec=.TRUE.
  CALL dseupd(rvec,'All',select,d,v,neq,sigma,bmat,neq,which,nev,tol,   &
             resid,ncv,v,neq,iparam,ipntr,workd,workl,lworkl,ierr)
!----- write out the spectrum -----
IF(ierr/=0 )THEN; WRITE(11,*)"Fatal error in sseupd"
  WRITE(11,*)"Should compute residuals"
END IF
WRITE(11,'(A )'"The eigenvalues are:"; WRITE(11,'(6E12.4 )')d(1:nev,1)
DO i=1,nev
  udiag(:)=v(1:neq,i); udiag=udiaig*diag
  WRITE(11,'(A,I3,A)' )" Eigenvector number",i," is:"
  WRITE(11,'(6E12.4 )')udiaig(1:)/MAXVAL(ABS(udiaig(1:)))
  IF(i==1)CALL dismsh((/zero,udiaig/),nf,0.1_iwp,g_coord,g_num,argv,    &
                     nlen,13)
END DO
END IF
STOP
END PROGRAM p104

```

The same 4-node problem considered previously is used to demonstrate this program with data as shown in Figure 10.12.

The output from the program is listed as Figure 10.13. It indicates the number of Arnoldi iterations (21 in this case) followed by the first five eigenvalues and corresponding eigenvectors. It can be noted that the results are the same as those obtained by Program 10.2 for the first example (Figure 10.5), which also used 4-node elements with lumped mass. This program uses ARPACK routines dsaupd and dseupd (see Lehoucq *et al.*, 1998 and Appendix G).

```

nxe  nyx  nod
      1     4

np_types
1

prop (e,v,rho)
1.0  0.0  1.0

etype (not needed)

x_coords, y_coords
0.0  0.8  1.6  2.4  3.2  4.0
0.0 -1.0

nr, (k,nf(:,k),i=1,nr)
2
1 0 0  2 0 0

nev  ncv  bmat  which  tol    maxitr
5     20    'I'     'SM'   1.E-4    2

```

**Figure 10.12** Data for Program 10.4 example

There are 20 equations

Convergence after 21 iterations

The eigenvalues are:

```
0.4595E-02 0.1053E+00 0.1529E+00 0.5169E+00 0.1226E+01
Eigenvector number 1 is:
0.7195E-01 -0.6932E-01 -0.7195E-01 -0.6932E-01 0.1224E+00 -0.2362E+00
-0.1224E+00 -0.2362E+00 0.1526E+00 -0.4662E+00 -0.1526E+00 -0.4662E+00
0.1662E+00 -0.7285E+00 -0.1662E+00 -0.7285E+00 0.1695E+00 -0.1000E+01
-0.1695E+00 -0.1000E+01
```

Eigenvector number 2 is:

```
0.2487E+00 -0.4240E+00 -0.2487E+00 -0.4240E+00 0.1022E+00 -0.8727E+00
-0.1022E+00 -0.8727E+00 -0.2584E+00 -0.7980E+00 0.2584E+00 -0.7980E+00
-0.5732E+00 -0.7556E-01 0.5732E+00 -0.7556E-01 -0.6870E+00 0.1000E+01
0.6870E+00 0.1000E+01
```

Eigenvector number 3 is:

```
0.3090E+00 0.4290E-15 0.3090E+00 0.4621E-15 0.5878E+00 0.1607E-14
0.5878E+00 0.1103E-14 0.8090E+00 0.1582E-15 0.8090E+00 0.2455E-15
0.9511E+00 -0.1971E-14 0.9511E+00 -0.2713E-14 0.1000E+01 -0.5659E-14
0.1000E+01 -0.5147E-14
```

Eigenvector number 4 is:

```
-0.2413E+00 0.9151E+00 0.2413E+00 0.9151E+00 0.4612E+00 0.8557E+00
-0.4612E+00 0.8557E+00 0.5141E+00 -0.3742E+00 -0.5141E+00 -0.3742E+00
-0.4120E+00 -0.6579E+00 0.4120E+00 -0.6579E+00 -0.1000E+01 0.7050E+00
0.1000E+01 0.7050E+00
```

Eigenvector number 5 is:

```
-0.2575E+00 -0.9693E+00 0.2575E+00 -0.9693E+00 -0.8105E+00 0.2304E+00
0.8105E+00 0.2304E+00 0.3806E+00 0.5583E+00 -0.3806E+00 0.5583E+00
0.6787E-01 -0.6923E+00 -0.6787E-01 -0.6923E+00 -0.1000E+01 0.2476E+00
0.1000E+01 0.2476E+00
```

**Figure 10.13** Results from Program 10.4 example

## 10.2 Glossary of Variable Names

### Scalar integers:

i	simple counters
idiag	skyline bandwidth
ido	see ARPACK documentation
iel	simple counters
info, irr	see ARPACK documentation
iflag	no start vector specified, set to -1
ishfts	see ARPACK documentation
iters	number of Lanczos iterations
iwp	SELECTED_REAL_KIND(15)
ifail	warning flag from bisect subroutine
j	simple counter
jflag	equals zero if eigenvectors computed properly
k	simple counter
lalfa	Lanczos iteration ceiling
leig	maximum number of eigenvalues in range
logfil	see ARPACK documentation
lp	unit number for diagnostic messages

lworkl	see ARPACK documentation
lx	set to at least $3 \times \text{leig}$
lz	holds first dimension of array z
maxitr	number of Arnoldi update iterations
model, msaitr, msapps,	see ARPACK documentation
msaupd, msaup2, mseigt,	
mseupd, msgsets	
nband	bandwidth of upper triangle
nconv	see ARPACK documentation
ncv	number of stored vectors for re-orthogonalisation
ndigit	see ARPACK documentation
ndim	bandwidth of upper triangle
nev	number of eigenvalues requested
ndof	number of degrees of freedom per element
neig	holds the number of eigenvalues in eig
nels	number of elements
neq	number of degrees of freedom in the mesh
nip	number of integrating points
nlen	maximum number of characters in data file basename
nmodes	number of eigenvectors required
nn	number of nodes
nod	number of nodes per element
nodof	number of degrees of freedom per node
nprops	number of material properties
np_types	number of different property types
nr	number of restrained nodes
nst	number of stress (strain) terms
nxe, nye	number of columns and rows of elements

**Scalar reals:**

acc	convergence tolerance relative to largest eigenvalue
area	element area
det	determinant of the Jacobian matrix
d12	set to 12.0
el	lower limit of eigenvalue spectrum
er	upper limit of eigenvalue spectrum
etol	eigenvalue tolerance set to $1 \times 10^{-30}$
one	set to 1.0
penalty	set to $1 \times 10^{20}$
pt5	set to 0.5
sigma	see ARPACK documentation
tol	convergence criterion
zero	set to 0.0

**Scalar character:**

argv	holds data file basename
bmat	nature of problem. For standard equals 'I'

element type  
which portion of spectrum requested. For smallest equals  
'SM'

**Scalar logical:**

rvec see ARPACK documentation

**Dynamic integer arrays:**

etyp	element property type vector
g	element steering vector
g_g	global element steering matrix
g_num	node numbers for all elements
iparam(11), ipntr(11)	see ARPACK documentation
jeig	used to get the eigenvectors
nf	nodal freedom matrix
nu	working array
num	element node number vector

**Dynamic real arrays:**

alfa	working array holding Lanczos tridiagonal matrix
bee	strain-displacement matrix
beta	working array holding Lanczos tridiagonal matrix
coord	element nodal coordinates
d	see ARPACK documentation
dee	stress-strain matrix
del	working array
der	shape function derivatives with respect to local coordinates
deriv	shape function derivatives with respect to global coordinates
diag	global lumped mass vector
ecm	Gauss point contribution to consistent mass matrix
eig	holds the computed eigenvalues in increasing order
ell	element lengths vector
fun	shape functions
g_coord	nodal coordinates for all elements
jac	Jacobian matrix
kb	global stiffness matrix
kdiag	diagonal term location vector
kh, km	element stiffness matrices
ku, kv	global stiffness matrices
mb	global mass matrix
mm	element mass or lumped mass matrix
pmul	used in element-by-element products
points	integrating point local coordinates
prop	element properties matrix
resid	see ARPACK documentation
rrmass	reciprocal square rooted global lumped mass matrix

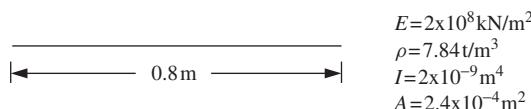
ua	working space vector
udiag	working space vector or eigenvector (in Lanczos)
utemp	used in element-by-element products
v	see ARPACK documentation
va	working space vector
vtemp	used in element-by-element products
v_store	holds the Lanczos vectors
weights	weighting coefficients
workd, workl	see ARPACK documentation
w1, x	working space vectors
x_coords	x-coordinates of mesh layout
y	working space vector
y_coords	y-coordinates of mesh layout
z	working space vector

**Dynamic logical arrays:**

select see ARPACK documentation

### 10.3 Exercises

1. Use Program 10.1 to evaluate the lowest two natural frequencies of the beam shown in Figure 10.14 with the boundary conditions:
- Pinned at both ends.
  - Fixed at both ends.
  - Fixed at one end and pinned at the other.
  - Fixed at one end and free at the other.

**Figure 10.14**

Answer: Using eight beam elements of equal length, ‘exact’ solutions in parentheses:

- $\omega_1 = 223(225)\text{s}^{-1}$ ,  $\omega_2 = 877(899)\text{s}^{-1}$
- $\omega_1 = 506(510)\text{s}^{-1}$ ,  $\omega_2 = 1364(1405)\text{s}^{-1}$
- $\omega_1 = 349(351)\text{s}^{-1}$ ,  $\omega_2 = 1107(1138)\text{s}^{-1}$
- $\omega_1 = 79(80)\text{s}^{-1}$ ,  $\omega_2 = 480(502)\text{s}^{-1}$

2. Repeat question 1 using Program 10.2 with 8-node elements.

Answer: Using a row of eight square ( $0.1 \times 0.1$ ) 8-node elements with  $E = 4800.0$ ,  $\nu = 0.0$  and  $\rho = 0.01882$ , ‘exact’ solutions in parentheses:

- $\omega_1 = 219(225) \text{ s}^{-1}$ ,  $\omega_2 = 824(899)\text{s}^{-1}$
- $\omega_1 = 474(510) \text{ s}^{-1}$ ,  $\omega_2 = 1196(1405)\text{s}^{-1}$
- $\omega_1 = 335(351) \text{ s}^{-1}$ ,  $\omega_2 = 1007(1138)\text{s}^{-1}$
- $\omega_1 = 80(80) \text{ s}^{-1}$ ,  $\omega_2 = 462(502)\text{s}^{-1}$

3. Use Program 10.2 with 4-node elements to estimate the first two axial natural frequencies and mode shapes of the rod shown in Figure 10.15.

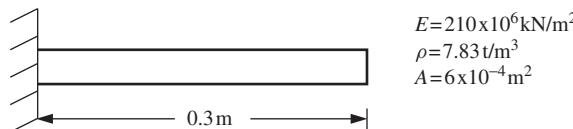


Figure 10.15

Answer: Using a row of five square ( $0.06 \times 0.06$ ) 4-node elements with  $E = 2.1 \times 10^6$ ,  $\nu = 0.0$  and  $\rho = 0.0783$ , ‘exact’ solutions in parentheses. Axial deformations appear in the third and sixth modes:

$$\omega_3 = 2.70 \times 10^4 (2.70 \times 10^4) \text{ s}^{-1}, [0.0 \ 0.31 \ 0.59 \ 0.81 \ 0.95 \ 1.00]^T$$

$$\omega_6 = 7.84 \times 10^4 (8.10 \times 10^4) \text{ s}^{-1}, [0.0 \ 0.81 \ 0.95 \ 0.31 \ -0.59 \ -1.00]^T$$

4. Repeat question 3 using rod elements. Program 10.1 is easily modified to analyse axial vibrations of rods. Make the following changes:

- In the declarations, change `ndof=4` to `ndof=2` and `nodof=2` to `nodof=1`.
- In the `elements_2`: loop, delete the statements that assign values to `mm(2, 2) = ...` and `mm(4, 4) = ...` and then change `mm(3, 3)` to `mm(2, 2)`.
- Change routine `beam_km` to `rod_km`.

The data read into `prop` will now be `ea` (instead of `ei`) and `rhoa`.

Answer: Using five equal rod elements with  $EA = 1.26 \times 10^5$  and  $\rho A = 4.70 \times 10^{-3}$  answers exactly the same as for question 3

5. Using the modified program from the previous question, determine the first two natural frequencies and eigenvectors for the stepped bar shown in Figure 10.16.

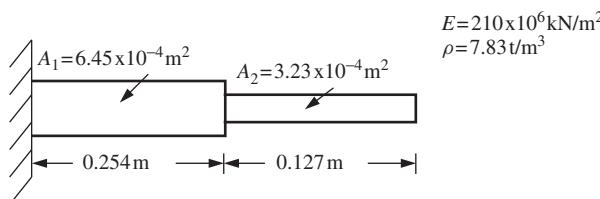


Figure 10.16

$$\text{Answer: } \omega_1 = 2.46 \times 10^4 \text{ s}^{-1}, [0.0 \ 0.50 \ 0.82 \ 0.95 \ 1.00]^T$$

$$\omega_2 = 5.94 \times 10^4 \text{ s}^{-1}, [0.0 \ 0.68 \ -0.08 \ -0.74 \ -1.00]^T$$

6. A vital attribute of an element ‘stiffness matrix’ is that it should possess the right number of ‘rigid body modes’, i.e. zero eigenvalues of the stiffness matrix. Test the 2-node elastic rod element stiffness matrix and prove that it has only one zero eigenvalue, corresponding to one rigid body mode (translation). If the rod has length  $L$ , cross-sectional area  $A$ , modulus  $E$  and mass per unit length  $\rho$ , calculate its non-zero eigenvalue and hence natural frequency of free vibration assuming both lumped and consistent mass. Compare with the ‘exact’ value of  $\pi/L\sqrt{E/\rho}$ .

Answer: lumped:  $2/L\sqrt{E/\rho}$ , consistent:  $2\sqrt{3}/L\sqrt{E/\rho}$

7. Show how dynamic equilibrium of a multi-degree of freedom system vibrating at a resonant frequency leads to an equation of the form

$$[\mathbf{K}_m]\{\mathbf{X}\} = \omega^2[\mathbf{M}_m]\{\mathbf{X}\}$$

where  $[\mathbf{K}_m]$ ,  $[\mathbf{M}_m]$  = system stiffness and mass matrices,  $\{\mathbf{X}\}$  = displacement amplitudes,  $\omega^2$  = angular frequency. Describe a method for reducing this equation to a standard eigenvalue equation of the form

$$[\mathbf{A}]\{\mathbf{Z}\} = \omega^2\{\mathbf{Z}\}$$

where  $[\mathbf{A}]$  is symmetrical.

## References

- Chopra AK 1995 *Dynamics of Structures*. Prentice-Hall, Englewood Cliffs, NJ.
- Cook RD, Malkus DS, Plesha ME and Witt RJ 2002 *Concepts and Applications of Finite Element Analysis*, 4th edn. John Wiley & Sons, Chichester.
- HSL 2011 A collection of Fortran codes for large scale scientific computation. <http://www.hsl.rl.ac.uk>.
- Lehoucq RB, Sorensen DC and Yang C 1998 *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM Press, Philadelphia.
- Parlett BN and Reid JK 1981 Tracking the progress of the Lanczos algorithm for large symmetric eigenproblems. *IMA J Num Anal* **1**, 135–155.
- Smith IM 1977 Transient phenomena of offshore foundations. In *Numerical Methods in Offshore Engineering*. John Wiley & Sons, Chichester, chapter 14.
- Smith IM 1984 Adaptability of truly modular software. *Eng Comput* **1**(1), 25–35.
- Smith IM and Heshmati EE 1983 Use of a Lanczos algorithm in dynamic analysis of structures. *Earthquake Eng Struc Dyn* **11**(4), 585–588.

# 11

## Forced Vibrations

### 11.1 Introduction

In the previous chapter, programs were described which enable the calculation of the intrinsic dynamic properties of systems, namely their undamped natural frequencies and mode shapes. The next stage in a dynamic analysis is usually the calculation of the response of the system to an imposed time-dependent disturbance. This chapter describes eight programs which enable such calculations to be made.

The types of equations to be solved were derived early in the book [e.g., (2.13)]. After semi-discretisation in space using finite elements, the resulting matrix equations are typified by (2.17), a set of second-order ordinary differential equations in time. On inclusion of damping and forcing terms, the relevant equations become (3.124) and Section 3.13 describes the principles behind the various solution procedures used below.

Program 11.1 describes forced vibration analysis of elastic slender beam structures using direct integration in the ‘time domain’. Programs 11.2, 11.3, 11.4 and 11.5 describe forced vibration analyses of planar 2D elastic solids. Program 11.2 works in the ‘frequency domain’ using the modal superposition method. Programs 11.3 and 11.4 work in the ‘time domain’ utilising two different implicit time-marching algorithms and Program 11.5 uses the complex response method. Program 11.6 illustrates a ‘mixed’ time-marching scheme for 2D analysis in which some parts of the mesh are integrated ‘explicitly’ and others ‘implicitly’. Program 11.7 repeats the algorithm of Program 11.3 using an ‘element-by-element’ approach with a preconditioned conjugate gradient solver. Program 11.8 uses a fully explicit time-marching scheme to analyse a 2D elastic–plastic material.

Programs 11.7 and 11.8 have parallel counterparts described in Chapter 12.

### Program 11.1 Forced vibration analysis of elastic beams using 2-node beam elements. Consistent mass. Newmark time stepping

```
PROGRAM p111
!-----
! Program 11.1 Forced vibration analysis of elastic beams using 2-node
!           beam elements. Consistent mass. Newmark time stepping.
!-----
```

```

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,j,k,lnode,lsense,ndof=4,nels,neq,nlen,nlf,nn,nod=2,      &
nodof=2,nof,nlfp,nprops=2,np_types,nr,nstep
REAL(iwp)::beta,dtim,fk,fm,f1,f2,gamma,one=1.0_iwp,pt5=0.5_iwp,      &
zero=0.0_iwp; CHARACTER(LEN=15)::argv
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g(:, :,kdiag(:,lf(:,lp(:,nf(:, :,&
node(:,num(:,sense(:)
REAL(iwp),ALLOCATABLE::a(:,acc(:, :,al(:, :,b1(:,cv(:,d(:,      &
dis(:, :,ell(:,kd(:,km(:, :,kp(:,kv(:,mc(:,mm(:, :,mv(:,      &
prop(:, :,rl(:,rt(:,v(:,vc(:,vel(:, :
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nels,np_types; nn=nels+1
ALLOCATE(nf(nodof,nn),km(ndof,ndof),mm(ndof,ndof),num(nod),g(ndof),      &
prop(nprops,np_types),ell(nels),g_g(ndof,nels),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)ell,dtim,beta,gamma,fm,fk
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag(neq),a1(0:neq),b1(0:neq),vc(0:neq),kd(0:neq),a(0:neq),      &
d(0:neq),v(0:neq)); kdiag=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
  num=(iel,iel+1); CALL num_to_g(num,nf,g); g_g(:,iel)=g
  CALL fkdiag(kdiag,g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),cv(kdiag(neq)),mv(kdiag(neq)),mc(kdiag(neq)),      &
kp(kdiag(neq)))
WRITE(11,'(2(A,I5))')
  " There are",neq," equations and the skyline storage is",kdiag(neq)
kv=zero; mv=zero
!-----global stiffness and mass matrix assembly-----
elements_2: DO iel=1,nels
  CALL beam_km(km,prop(1,etype(iel)),ell(iel))
  CALL beam_mm(mm,prop(2,etype(iel)),ell(iel)); g=g_g(:,iel)
  CALL fsparv(kv,km,g,kdiag); CALL fsparv(mv,mm,g,kdiag)
END DO elements_2; mc=mv
!-----initial conditions, and load functions-----
d=zero; v=zero ! alternatively READ(10,*)d(1:),v(1:)
READ(10,*)nlf; ALLOCATE(lf(nlf))
DO k=1,nlf
  READ(10,*)lnode,lsense,nlfp; ALLOCATE(rt(nlfp),rl(nlfp))
  lf(k)=nf(lsense,lnode); READ(10,*)(rt(j),rl(j),j=1,nlfp)
  IF(k==1)THEN
    nstep=NINT((rt(nlfp)-rt(1))/dtim)+1; ALLOCATE(al(nstep,nlf))
  END IF; CALL interp(k,dtim,rt,rl,al,nstep); DEALLOCATE(rt,rl)
END DO
f1=beta*dtim**2; f2=beta*dtim; cv=fm*mv+fk*kv; kp=mv/f1+gamma*cv/f2+kv
CALL sparin(mc,kdiag); CALL sparin(kp,kdiag); a=zero; a(lf(:))=al(1,:)
CALL linmul_sky(cv,v,vc,kdiag); CALL linmul_sky(kv,d,kd,kdiag); a=a-vc-kd
CALL spabac(mc,a,kdiag); READ(10,*)nof

```

```

ALLOCATE(node(nof), sense(nof), lp(nof), dis(nstep,nof), vel(nstep,nof),      &
acc(nstep,nof)); READ(10,*) (node(i),sense(i),i=1,nof)
DO i=1,nof; lp(i)=nf(sense(i),node(i)); END DO
dis(1,:)=d(lp); vel(1,:)=v(lp); acc(1,:)=a(lp)
!-----time stepping loop-----
DO j=2,nstep
  a1=d/f1+v/f2+a*(pt5/beta-one)
  b1=gamma*d/f2-v*(one-gamma/beta)-dtim*a*(one-pt5*gamma/beta)
  CALL linmul_sky(mv,a1,vc,kdiag); CALL linmul_sky(cv,b1,kd,kdiag)
  d=vc+kd; d(lf(:))=d(lf(:))+a1(j,:); CALL spabac(kp,d,kdiag)
  v=gamma*d/f2-b1; a=d/f1-a1
  dis(j,:)=d(lp); vel(j,:)=v(lp); acc(j,:)=a(lp)
END DO
DO i=1,nof
  WRITE(11,'(/,2(A,I3))')" Output at node",node(i)," , sense",sense(i)
  WRITE(11,'(A)')" time disp velo accel"
  DO j=1,nstep; WRITE(11,'(4E12.4)')(j-1)*dtim,dis(j,i),vel(j,i),acc(j,i)
  END DO
END DO
STOP
END PROGRAM p111

```

This program computes the response of a string of beam elements to a combination of time-dependent applied nodal loads. The program allows for (Rayleigh) damping (Section 3.13) and uses a consistent mass matrix (2.30). Time stepping is achieved using a direct Newmark method involving time-stepping parameters  $\beta$  and  $\gamma$ , the values of which determine the accuracy and stability characteristics of the algorithm (see, e.g., Bathe, 1996). For the special case of  $\beta = 1/4$  and  $\gamma = 1/2$ , the method is identical to the Crank–Nicolson  $\theta = 0.5$  method described in Section 3.13.2.

Starting from the assembled form of (3.124), and using a ‘dot’ notation to signify time derivatives, we have

$$[\mathbf{K}_m]\{\mathbf{U}\} + [\mathbf{C}_m]\{\dot{\mathbf{U}}\} + [\mathbf{M}_m]\{\ddot{\mathbf{U}}\} = \{\mathbf{F}\} \quad (11.1)$$

with known initial conditions on displacements and velocities,  $\{\mathbf{U}\}_0$  and  $\{\dot{\mathbf{U}}\}_0$ .

Let  $\{\mathbf{F}\}_i$ ,  $\{\mathbf{U}\}_i$ ,  $\{\dot{\mathbf{U}}\}_i$  and  $\{\ddot{\mathbf{U}}\}_i$  represent conditions at time  $t = i\Delta t$ , where  $i = 0, 1, 2, \dots, n_{\text{step}}$ .

Assuming that  $[\mathbf{M}_m]$ ,  $[\mathbf{C}_m]$ ,  $[\mathbf{K}_m]$ ,  $\beta$ ,  $\gamma$  and  $\Delta t$  are constant, and that  $\{\mathbf{F}\}_i$  is known for all  $i$ , the following algorithm is used to obtain the values of  $\{\mathbf{U}\}_i$ ,  $\{\dot{\mathbf{U}}\}_i$  and  $\{\ddot{\mathbf{U}}\}_i$  for all  $i > 0$ .

(1) Compute

$$[\mathbf{K}'] = [\mathbf{K}_m] + \frac{\gamma}{\beta\Delta t}[\mathbf{C}] + \frac{1}{\beta(\Delta t)^2}[\mathbf{M}_m]$$

(2) Factorise  $[\mathbf{K}']$  to facilitate step 8

(3) Solve the linear equations  $[\mathbf{M}_m]\{\ddot{\mathbf{U}}\}_0 = \{\mathbf{F}\}_0 - [\mathbf{C}_m]\{\dot{\mathbf{U}}\}_0 - [\mathbf{K}_m]\{\mathbf{U}\}_0$

(4) Set  $i = 0$

(5) Compute

$$\{\mathbf{A}\}_i = \frac{1}{\beta(\Delta t)^2} \{\mathbf{U}\}_i + \frac{1}{\beta \Delta t} \{\dot{\mathbf{U}}\}_i + \left( \frac{1}{2\beta} - 1 \right) \{\ddot{\mathbf{U}}\}_i$$

(6) Compute

$$\{\mathbf{B}\}_i = \frac{\gamma}{\beta \Delta t} \{\mathbf{U}\}_i - \left( 1 - \frac{\gamma}{\beta} \right) \{\dot{\mathbf{U}}\}_i - \left( 1 - \frac{\gamma}{2\beta} \right) \Delta t \{\ddot{\mathbf{U}}\}_i$$

(7) Compute  $\{\mathbf{F}'\}_{i+1} = \{\mathbf{F}\}_{i+1} + [\mathbf{M}_m]\{\mathbf{A}\}_i + [\mathbf{C}_m]\{\mathbf{B}\}_i$

(8) Solve the linear equations  $[\mathbf{K}']\{\mathbf{U}\}_{i+1} = \{\mathbf{F}'\}_{i+1}$

(9) Compute

$$\{\dot{\mathbf{U}}\}_{i+1} = \frac{\gamma}{\beta \Delta t} \{\mathbf{U}\}_{i+1} - \{\mathbf{B}\}_i$$

(10) Compute

$$\{\ddot{\mathbf{U}}\}_{i+1} = \frac{1}{\beta(\Delta t)^2} \{\mathbf{U}\}_{i+1} - \{\mathbf{A}\}_i$$

(11) Increment  $i$  and repeat from step 5

Subroutine `beam_mm` (see, e.g., Program 4.3) forms the beam element consistent mass matrix and subroutine `interp` takes the input load/time function data points and interpolates linearly to give load/time function values at the resolution of the calculation time step.

The example and data shown in Figure 11.1 are of a cantilever of unit length modelled with a single beam element, subjected to a tip loading given by a half-sine pulse with an amplitude of  $EI$  and a duration equal to  $T$ , the fundamental period of the cantilever. The properties of the beam are given by an  $EI$  of 3.194 and a mass per unit length  $\rho A$  of 1.0. The fundamental frequency of the beam from equation (10.4) is therefore given by  $\omega = 3.516\sqrt{3.194} = 6.284$ , and the fundamental period by  $T = 2\pi/\omega = 1.00$ .

The first line of data reads the number of elements `nels` followed by the number of property groups and the properties  $EI$  and  $\rho A$ . The next line reads the element lengths `e11` and the next line gives the time-stepping data. In this case the calculation time step is `dtime=0.05` and the conventional Newmark time-stepping parameters are used, namely  $\beta = 0.5$  and  $\gamma = 0.25$  (read as `beta` and `gamma`, respectively). The beam is undamped, thus the Rayleigh damping parameters (`fm` and `fk`) are both set to zero. The boundary condition data indicate one restrained node at the built-in end of the cantilever, where both freedoms are restrained. There is one loaded freedom (`nlf=1`) in this example at node 2, sense 1 (the translational freedom). The sinusoidal loading function to be applied has been input using a total of 12 coordinates, comprising 11 coordinates at 0.1 s intervals up to 1 s, followed by a ‘quiet zone’ involving a single interval of 0.8 s up to 1.8 s. The program linearly interpolates this load/time function at the calculation step length of 0.05 s for the Newmark algorithm. The final data gives the freedoms at which output is required. In this example, the displacement, velocity and acceleration under the load are of interest, so there is just one output required (`nof=1`), again at node 2 sense 1.

The output from Program 11.1 is shown in Figure 11.2 and a plot of the computed displacement/time history of the cantilever tip is shown in Figure 11.3. For comparison,

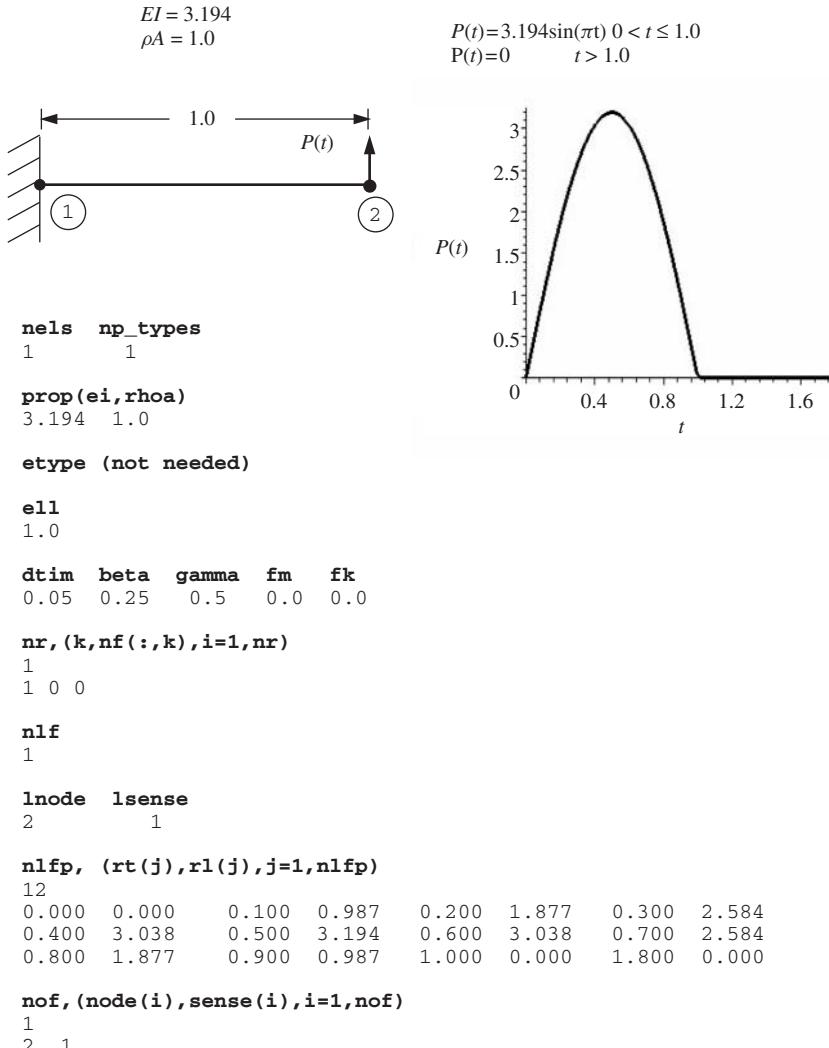


Figure 11.1 Mesh and data for Program 11.1 example

the analytical solution (e.g., Warburton, 1964) for this problem during the loading phase when  $0 \leq t \leq T$  is given by

$$v(t) = 0.441 \sin \frac{\pi t}{T} - 0.216 \sin \frac{2\pi t}{T} \quad (11.2)$$

and for the free vibration phase when  $t > T$  by

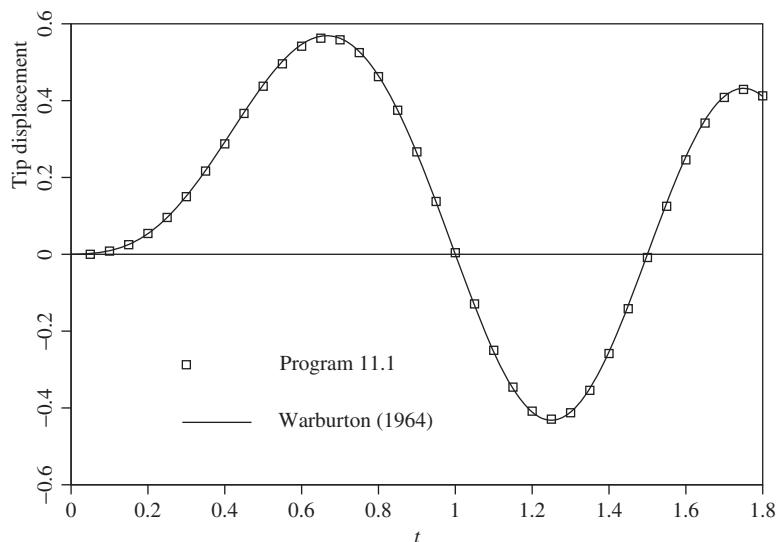
$$v(t) = -0.432 \sin\{6.284(t - T)\} \quad (11.3)$$

It is clear from Figure 11.3 that this analytical solution is virtually indistinguishable from the computed result.

There are 2 equations and the skyline storage is 3

```
Output at node 2, sense 1
      time      disp      velo      accel
0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
0.5000E-01  0.1942E-02  0.7768E-01  0.3107E+01
0.1000E+00  0.9511E-02  0.2251E+00  0.2788E+01
0.1500E+00  0.2531E-01  0.4069E+00  0.4485E+01
0.2000E+00  0.5285E-01  0.6946E+00  0.7026E+01
0.2500E+00  0.9488E-01  0.9866E+00  0.4652E+01
0.3000E+00  0.1497E+00  0.1207E+01  0.4159E+01
0.3500E+00  0.2153E+00  0.1415E+01  0.4184E+01
0.4000E+00  0.2892E+00  0.1541E+01  0.8437E+00
0.4500E+00  0.3650E+00  0.1493E+01  -0.2780E+01
0.5000E+00  0.4362E+00  0.1352E+01  -0.2843E+01
0.5500E+00  0.4976E+00  0.1104E+01  -0.7080E+01
0.6000E+00  0.5421E+00  0.6758E+00  -0.1005E+02
0.6500E+00  0.5633E+00  0.1746E+00  -0.9996E+01
0.7000E+00  0.5589E+00  -0.3524E+00  -0.1108E+02
0.7500E+00  0.5261E+00  -0.9574E+00  -0.1312E+02
0.8000E+00  0.4638E+00  -0.1534E+01  -0.9945E+01
0.8500E+00  0.3757E+00  -0.1993E+01  -0.8412E+01
0.9000E+00  0.2659E+00  -0.2397E+01  -0.7734E+01
0.9500E+00  0.1393E+00  -0.2670E+01  -0.3187E+01
0.1000E+01  0.4299E-02  -0.2729E+01  0.8097E+00
0.1050E+01  -0.1284E+00  -0.2581E+01  0.5126E+01
0.1100E+01  -0.2487E+00  -0.2229E+01  0.8928E+01
0.1150E+01  -0.3455E+00  -0.1642E+01  0.1458E+02
0.1200E+01  -0.4081E+00  -0.8625E+00  0.1658E+02
0.1250E+01  -0.4308E+00  -0.4551E-01  0.1610E+02
0.1300E+01  -0.4123E+00  0.7824E+00  0.1701E+02
0.1350E+01  -0.3534E+00  0.1575E+01  0.1468E+02
0.1400E+01  -0.2597E+00  0.2174E+01  0.9284E+01
0.1450E+01  -0.1415E+00  0.2554E+01  0.5935E+01
0.1500E+01  -0.9326E-02  0.2733E+01  0.1196E+01
0.1550E+01  0.1243E+00  0.2614E+01  -0.5935E+01
0.1600E+01  0.2452E+00  0.2221E+01  -0.9787E+01
0.1650E+01  0.3422E+00  0.1660E+01  -0.1265E+02
0.1700E+01  0.4067E+00  0.9181E+00  -0.1702E+02
0.1750E+01  0.4310E+00  0.5512E-01  -0.1750E+02
0.1800E+01  0.4132E+00  -0.7674E+00  -0.1540E+02
```

**Figure 11.2** Results from Program 11.1 example



**Figure 11.3** Computed tip displacement–time history from Program 11.1 example

## 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

```

PROGRAM p112
!-----
! Program 11.2 Forced vibration of an elastic solid in plane strain
!           using 4- or 8-node rectangular quadrilaterals. Lumped mass.
!           Mesh numbered in x- or y-direction. Modal superposition.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,idiag,iel,ifail,j,jj,k,nband,ndim=2,ndof,nels,neq,nlen,nmodes,&
nn,nod,nodof=2,npri,nprops=3,np_types,nr,nres,nstep,nxe,nye
REAL(iwp)::aa,area,bb,dr,dtim,d4=4.0_iwp,etol=1.e-30_iwp,f,k1,k2,omega,&
one=1.0_iwp,penalty=1.0e20_iwp,time,two=2.0_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g (:,:)),g_num(:, :,),kdiag(:,nf(:, :,)&
num(:))
REAL(iwp),ALLOCATABLE::bigk(:, :,),coord(:, :,),diag(:, g_coord(:, :,),kh(:, ),&
km(:, :,),ku(:, :,),kv(:, ),mm(:, :,),prop(:, :,),rrmass(:, ),udiag(:, ),xmod(:, ),&
x_coords(:, ),y_coords(:, )
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,nod,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ndof=nod*nodof
ALLOCATE(nf(ndof,nn),g_coord(ndim,nn),coord(nod,ndim),g_num(nod,nels), &
num(nod),km(ndof,ndof),g(ndof),g_g(ndof,nels),x_coords(nxe+1), &
y_coords(nye+1),prop(nprops,np_types),etype(nels),mm(ndof,ndof))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtim,nstep,npri,nres,dr
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
READ(10,*)nmodes,omega
ALLOCATE(diag(0:neq),udiag(0:neq),kdiag(neq),rrmass(0:neq),xmod(nmodes), &
bigk(neq,nmodes))
!-----loop the elements to find nband and set up global arrays-----
nband=0; kdiag=0
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
    IF(nband<bandwidth(g))nband=bandwidth(g)
END DO elements_1
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
WRITE(11,'(A,I5,A,/,A,I5,/,A,I5)')" There are",neq," equations", &
" The half-bandwidth (including diagonal) is",nband+1, &
" The skyline storage is",kdiag(neq)
diag=zero; bigk=zero
!-----element stiffness and mass assembly-----
ALLOCATE(ku(neq,nband+1),kv(kdiag(neq)),kh(kdiag(neq))); ku=zero
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    CALL rect_km(km,coord,prop(1,etype(iel)),prop(2,etype(iel)))

```

```

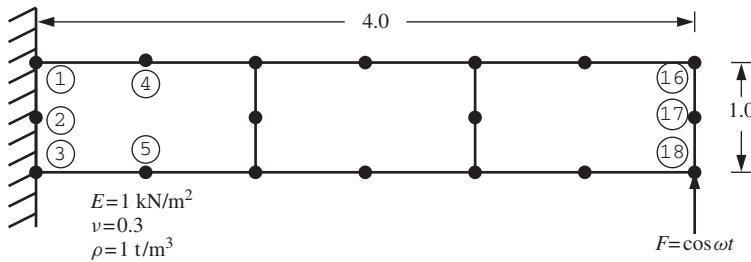
CALL formku(ku,km,g); area=(MAXVAL(coord(:,1))-MINVAL(coord(:,1)))*      &
      (MAXVAL(coord(:,2))-MINVAL(coord(:,2)))
CALL elmat(area,prop(3,etype(iel)),mm); CALL formlump(diag,mm,g)
END DO elements_2; rrmass(1:)=one/SQRT(diag(1:))
!-----reduce to standard eigenvalue problem-----
DO i=1,neq; IF(i<=neq-nband)THEN; k=nband+1; ELSE; k=neq-i+1; END IF
    DO j=1,k; ku(i,j)=ku(i,j)*rrmass(i)*rrmass(i+j-1); END DO
END DO
!-----convert to skyline form-----
kh(1)=ku(1,1); k=1
DO i=2,neq; iddiag=kdiag(i)-kdiag(i-1)
    DO j=1,iddiag; k=k+1; kh(k)=ku(i+j-iddiag,1-j+iddiag); END DO
END DO
!-----extract the eigenvalues-----
CALL banded(ku,diag,udiag); ifail=1; CALL bisect(diag,udiag,etol,ifail)
!-----extract the "mass normalised" eigenvectors-----
DO i=1,nmodes
    kv=kh; kv(kdiag)=kv(kdiag)-diag(i); kv(1)=kv(1)+penalty
    udiag=zero; udiag(1)=kv(1)
    CALL sparin_gauss(kv,kdiag); CALL spabac_gauss(kv,udiag,kdiag)
    udiag=rrmass*udiag
    udiag(1:)=udiag(1:)/SQRT(SUM(udiag(1:)**2/rrmass(1:)**2))
    bigk(:,i)=udiag(1:)
END DO; udiag=zero; time=zero
WRITE(11,'(/A,I5)')" Result at node",nres
WRITE(11,'(A)')"      time          load          x-disp          y-disp"
WRITE(11,'(4E12.4)')time,COS(omega*time),udiag(:,nres))
!-----time stepping loop-----
DO jj=1,nstep; time=time+dtime
    DO i=1,nmodes; f=bigk(neq,i)
!-----analytical solution for cosine loading-----
        k1=diag(i)-omega**2; k2=k1*k1+d4*omega**2*drt**2*diag(i)
        aa=f*k1/k2; bb=f*two*omega*drt*SQRT(diag(i))/k2
        xmod(i)=aa*COS(omega*time)+bb*SIN(omega*time)
    END DO
!-----superpose the modes-----
    udiag(1:)=MATMUL(bigk,xmod(1:))
    IF(jj/npri*npri==jj)WRITE(11,'(4E12.4)')time,COS(omega*time),      &
        udiag(:,nres))
END DO
STOP
END program p112

```

Since the basis of this method is the synthesis of the undamped natural modes of the vibrating system, it follows very naturally from the programs of the previous chapter. Indeed this program can be built up, with minor extensions, from Program 10.2. The method is described in Section 3.13.1.

The illustrative problem chosen for this program and the three that follow is shown in Figure 11.4 and represents a beam subjected to a harmonic vertical force of  $\cos \omega t$  at node 18. The damping ratio  $\gamma$  [see equation (3.126)], called  $drt$  in the program, is 0.05 or 5% applied to all modes of the system.

The forcing frequency  $\omega$  (called  $\omega$  in the program) is set at 0.3, which is deliberately chosen to be close to the second natural frequency of the undamped system.



```

nx e   ny e   nod
3      1     8

np_types
1

prop(e,v,rho)
1.0  0.3  1.0

etyp e (not needed)

x_coords, y_coords
0.0 1.33333 2.66667 4.0
0.0 -1.0

dtim nstep npri nres dr
1.0    20    1    18  0.05

nr,(k,nf(:,k),i=1,nr)
3
1 0 0 2 0 0 3 0 0

nmodes omega
6          0.3

```

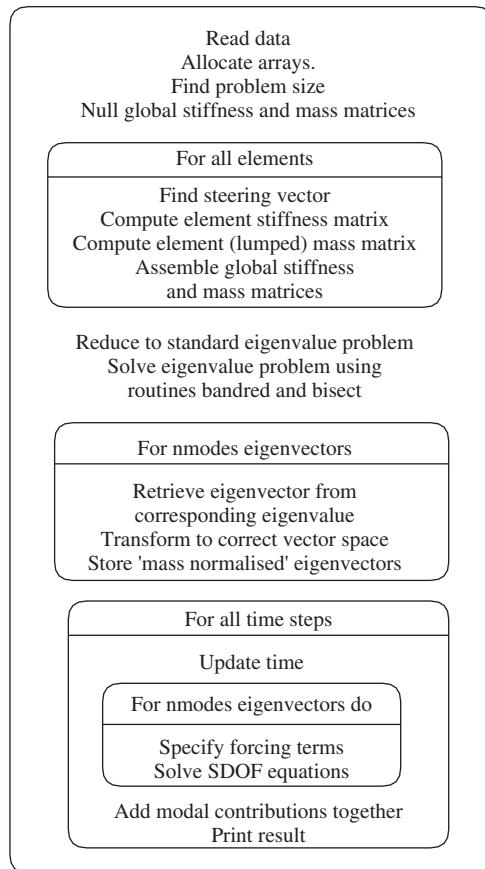
**Figure 11.4** Mesh and data for Program 11.2 example

(Note:  $\omega_2 = \sqrt{0.09363} \approx 0.3$  from Figure 10.7), so at this frequency the influence of damping should be significant.

The structure of the program is essentially the same as Program 10.2 up to the end of the section headed ‘extract the mass normalised eigenvectors’, however the current program uses a ‘Euclidean’ normalisation strategy that reduces the modal mass matrix to a unit matrix. It should be remembered that the eigenvectors first computed as *udiag* are those of the transformed problem and the true eigenvectors must be recovered prior to normalisation and storage in the eigenvector matrix *bigk*.

When the time-stepping loop is entered, the cosine loading of a single degree of freedom system is introduced, and the modal contributions superposed at each degree of freedom. A structure chart for the modal superposition algorithm is given in Figure 11.5.

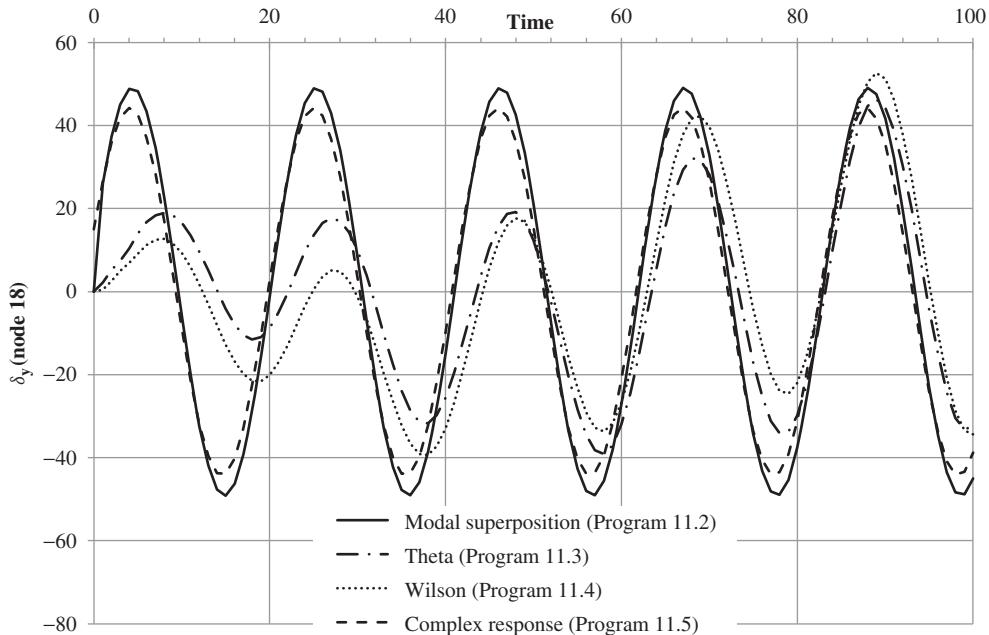
The example uses the first six eigenmodes, read as *nmodes*=6, to synthesise the time response. Users are invited to examine the sensitivity of the response to different values of *nmodes* up to a maximum of 30 in this case. The output from the analysis is shown in Figure 11.6 and the response in the *y*-direction at node 18 is plotted in Figure 11.7.

**Figure 11.5** Structure chart for Program 11.2

There are 30 equations and the half-bandwidth is 15

Result at node	18	time	load	x-disp	y-disp
0.0000E+00	0.1000E+01	0.0000E+00	0.0000E+00		
0.1000E+01	0.9553E+00	0.2188E+02	0.2612E+02		
0.2000E+01	0.8253E+00	0.2825E+02	0.3727E+02		
0.3000E+01	0.6216E+00	0.3210E+02	0.4508E+02		
0.4000E+01	0.3624E+00	0.3308E+02	0.4887E+02		
0.5000E+01	0.7074E-01	0.3110E+02	0.4829E+02		
0.6000E+01	-0.2272E+00	0.2635E+02	0.4340E+02		
0.7000E+01	-0.5048E+00	0.1924E+02	0.3463E+02		
0.8000E+01	-0.7374E+00	0.1042E+02	0.2277E+02		
0.9000E+01	-0.9041E+00	0.6594E+00	0.8875E+01		
0.1000E+02	-0.9900E+00	-0.9155E+01	-0.5813E+01		
0.1100E+02	-0.9875E+00	-0.1815E+02	-0.1998E+02		
0.1200E+02	-0.8968E+00	-0.2553E+02	-0.3237E+02		
0.1300E+02	-0.7259E+00	-0.3062E+02	-0.4186E+02		
0.1400E+02	-0.4903E+00	-0.3298E+02	-0.4761E+02		
0.1500E+02	-0.2108E+00	-0.3240E+02	-0.4911E+02		
0.1600E+02	0.8750E-01	-0.2892E+02	-0.4623E+02		
0.1700E+02	0.3780E+00	-0.2285E+02	-0.3921E+02		
0.1800E+02	0.6347E+00	-0.1475E+02	-0.2869E+02		
0.1900E+02	0.8347E+00	-0.5325E+01	-0.1561E+02		
0.2000E+02	0.9602E+00	0.4572E+01	-0.1135E+01		

**Figure 11.6** Results from Program 11.2 example



**Figure 11.7** Cantilever tip displacement computed by Programs 11.2, 11.3, 11.4 and 11.5

**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**

```
PROGRAM p113
!-----
! Program 11.3 Forced vibration analysis of an elastic solid in plane
! strain using rectangular 8-node quadrilaterals. Lumped or
! consistent mass. Mesh numbered in x- or y-direction.
! Implicit time integration using the "theta" method.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=9,nlen,nn,   &
nod=8,nodof=2,npri,nprops=3,np_types,nr,nres,nst=3,nstep,nxe,nye
REAL(iwp)::area,c1,c2,c3,c4,det,dtim,fk,fm,one=1.0_iwp,theta,time,      &
zero=0.0_iwp; LOGICAL::consistent=.FALSE.
CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:),g_g(:,:,),g_num(:,:,),kdiag(:,),nf(:,:,),  &
node(:),num(:)
REAL(iwp),ALLOCATABLE::bee(:,:),coord(:,:),dee(:,:),der(:,:),deriv(:,:),
d1x0(:),d1x1(:),d2x0(:),d2x1(:),ecm(:,:,),fun(:),f1(:),g_coord(:,:,),  &
```

```

jac(:,:,km(:,:,kv(:),loads(:,:,mm(:,:,mv(:),points(:,:,prop(:,:,&
val(:,:,weights(:),x0(:),x1(:),x_coords(:),y_coords(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn), &
dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),&
deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),num(nod),g_num(nod,nels), &
g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),fun(nod),etype(nels), &
prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtim,nstep,theta,npri,nres,fm,fk
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(x0(0:neq),d1x0(0:neq),x1(0:neq),d2x0(0:neq),loads(0:neq), &
d1x1(0:neq),d2x1(0:neq),kdiag(neq))
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
CALL sample(element,points,weights); kdiag=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nel
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),mv(kdiag(neq)),f1(kdiag(neq)))
WRITE(11,'(2(A,I5))') &
" There are",neq," equations and the skyline storage is",kdiag(neq)
kv=zero; mv=zero
!-----global stiffness and mass matrix assembly-----
elements_2: DO iel=1,nel
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    g=g_g(:,iel); km=zero; mm=zero; area=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        area=area+det*weights(i)
        IF(consistent)THEN; CALL ecmat(ecm,fun,ndof,nodof)
            mm=mm+ecm*det*weights(i)*prop(3,etype(iel))
        END IF
    END DO gauss_pts_1
    IF(.NOT.consistent)CALL elmat(area,prop(3,etype(iel)),mm)
    CALL fsparv(kv,km,g,kdiag); CALL fsparv(mv,mm,g,kdiag)
END DO elements_2
!-----initial conditions and factorise equations-----
x0=zero; d1x0=zero; d2x0=zero
c1=(one-theta)*dtim; c2=fk-c1; c3=fm+one/(theta*dtim); c4=fk+theta*dtim
f1=c3*mv+c4*kv; CALL sparin(f1,kdiag); time=zero
!-----time stepping loop-----
WRITE(11,'(/A,I5))' " Result at node",nres
WRITE(11,'(A)')"      time      load      x-disp      y-disp"
WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))

```

```

timesteps: DO j=1,nstep
    time=time+dtim; loads=zeros; x1=c3*x0+d1x0/theta
    DO i=1,loaded_nodes
        loads(nf(:,node(i)))=
            val(i,:)*(theta*dtim*load(time)+c1*load(time-dtim)) &
    END DO; CALL linmul_sky(mv,x1,d1x1,kdiag); d1x1=loads+d1x1; loads=c2*x0
    CALL linmul_sky(kv,loads,x1,kdiag); x1=x1+d1x1; CALL spabac(f1,x1,kdiag)
    d1x1=(x1-x0)/(theta*dtim)-d1x0*(one-theta)/theta
    d2x1=(d1x1-d1x0)/(theta*dtim)-d2x0*(one-theta)/theta
    x0=x1; d1x0=d1x1; d2x0=d2x1
    IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))
    END DO timesteps
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!-----Load-time function-----
IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
REAL(iwp),INTENT(IN)::t
REAL(iwp)::load_result
load_result=COS(0.3_iwp*t)
RETURN
END FUNCTION load
END PROGRAM p113

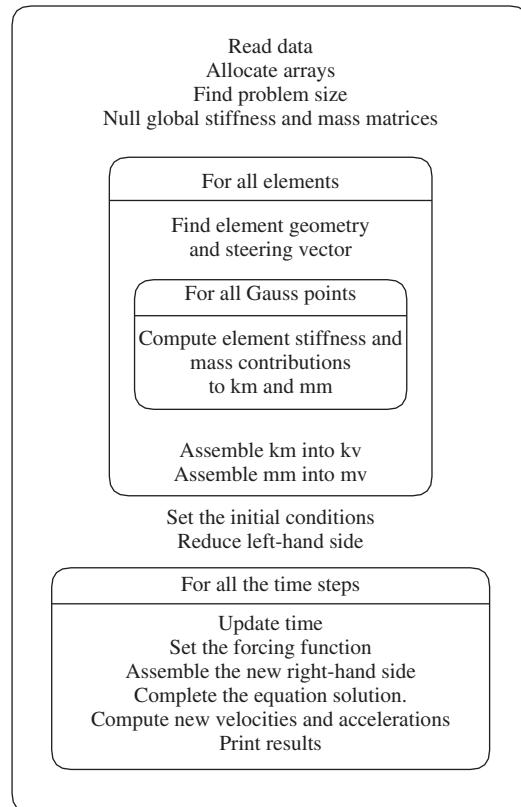
```

In this program whose structure chart is given as Figure 11.8, the same problem as that previously analysed is solved again using a direct time-integration procedure. In this example the tip loading takes the form of a continuous cosine function defined in a function subprogram called `load` placed at the end of the main program. The specific method is described in Section 3.13.2 where it was shown to be the same implicit technique as was used for first-order problems in Chapter 8, where it is often called the ‘Crank–Nicolson’ approach ( $\theta = 0.5$ ). In second-order problems, it is also known as the ‘Newmark  $\beta = 1/4$ ’ method in which form it was used in Program 11.1.

To step from one time instant to the next, a set of simultaneous equations has to be solved. Since the differential equations are often linearised, this is not as great a numerical task as might be supposed because the equation coefficients are constant, and need be factorised only once before the time-stepping procedure commences [see equation (3.145)]. Velocities and accelerations are computed by ancillary equations (3.146) and (3.147).

The example presented here uses lumped mass (`consistent = .FALSE.`), but the program can also handle consistent mass if required, in which case the element mass matrix is formed by subroutine `ecmat`. It should be noted that ‘exact’ integration is needed to exactly integrate the consistent mass matrix of a general 8-node quadrilateral element. In this example, therefore, nine integrating points (`nip=9`) have been used for all the element integrations. If, as is often the case, ‘reduced’ (`nip=4`) integration is preferred in the generation of the stiffness matrix, two separate integration loops would be required, one for the (consistent) mass and one for the stiffness.

Up to the section headed ‘global stiffness and mass matrix assembly’, the program’s task is the familiar one of generating the global stiffness and mass matrices, stored as usual as skyline vectors `kv` and `mv`, respectively. The matrix arising on the

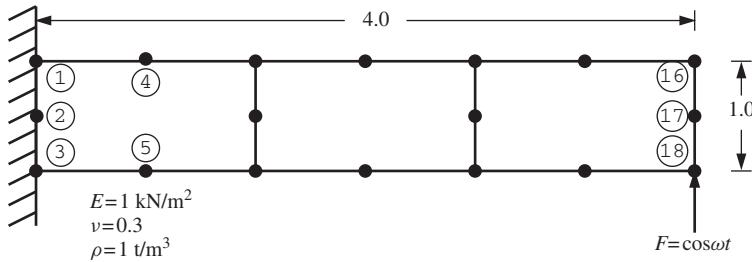


**Figure 11.8** Structure chart for implicit algorithms used in Programs 11.3 and 11.4

left-hand side of (3.145) is then created, called (*f1*) and factorised using subroutine *sparin*.

In the time-stepping loop, the matrix-by-vector multiplications and vector additions specified on the right-hand side of (3.145) are carried out and equation solution is completed by subroutine *spabac*. It then remains only to compute the new velocities and accelerations using (3.146) and (3.147).

The problem layout and data are given in Figure 11.9. Following the usual data relating to element numbers, properties and coordinates, the time-stepping data calls for *nstep*=20 steps of time step *dtim*=1.0 with the output at node *nres*=18 printed every *npri*=1 time step. The beam is damped using Rayleigh damping constants  $f_m = 0.005$  and  $f_k = 0.272$  (read as *fm* and *fk*, respectively), giving a damping ratio  $\gamma$  close to 0.05 [see equation (3.126)] for the first three natural frequencies. The results are listed as Figure 11.10 and plotted in Figure 11.7. The displacements are seen to be considerably disturbed by the initial condition ‘transients’, but once the response has settled down, the average amplitude agrees closely with that obtained by modal superposition, with a phase shift of about 90° relative to the loading function.



```

nx   ny
3     1

np_types
1

prop(e,v,rho)
1.0  0.3  1.0

etype (not needed)

x_coords, y_coords
0.0  1.33333  2.66667  4.0
0.0  -1.0

dtim  nstep  theta  npri  nres  fm    fk
1.0    20    0.5     1     18   0.005  0.272

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
1
18 0.0 1.0

```

**Figure 11.9** Mesh and data for Program 11.3 example

There are 30 equations and the skyline storage is 285

```

Result at node 18
time      load      x-disp      y-disp
0.0000E+00  0.1000E+01  0.0000E+00  0.0000E+00
0.1000E+01  0.9553E+00  0.7363E+00  0.2167E+01
0.2000E+01  0.8253E+00  0.2231E+01  0.5226E+01
0.3000E+01  0.6216E+00  0.3252E+01  0.7398E+01
0.4000E+01  0.3624E+00  0.3987E+01  0.1046E+02
0.5000E+01  0.7074E-01  0.4832E+01  0.1398E+02
0.6000E+01  -0.2272E+00  0.5272E+01  0.1678E+02
0.7000E+01  -0.5048E+00  0.5086E+01  0.1839E+02
0.8000E+01  -0.7374E+00  0.4370E+01  0.1894E+02
0.9000E+01  -0.9041E+00  0.3255E+01  0.1834E+02
0.1000E+02  -0.9900E+00  0.1789E+01  0.1650E+02
0.1100E+02  -0.9875E+00  0.8872E-01  0.1349E+02
0.1200E+02  -0.8968E+00  -0.1674E+01  0.9574E+01
0.1300E+02  -0.7259E+00  -0.3372E+01  0.5082E+01
0.1400E+02  -0.4903E+00  -0.4915E+01  0.3726E+00
0.1500E+02  -0.2108E+00  -0.6150E+01  -0.4128E+01
0.1600E+02  0.8750E-01  -0.6858E+01  -0.7908E+01
0.1700E+02  0.3780E+00  -0.6839E+01  -0.1050E+02
0.1800E+02  0.6347E+00  -0.5986E+01  -0.1157E+02
0.1900E+02  0.8347E+00  -0.4304E+01  -0.1096E+02
0.2000E+02  0.9602E+00  -0.1902E+01  -0.8708E+01

```

**Figure 11.10** Results from Program 11.3 example

**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**

```

PROGRAM p114
!-----
! Program 11.4 Forced vibration analysis of an elastic solid in plane
! strain using rectangular 8-node quadrilaterals. Lumped or
! consistent mass. Mesh numbered in x- or y-direction.
! Implicit time integration using Wilson's method.
!-----

USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=9,nlen,nn,      &
nod=8,nodof=2,npri,nprops=3,np_types,nr,nres,nst=3,nstep,nxe,nye
REAL(iwp)::area,c1,c2,c3,c4,det,dtim,d6=6.0_iwp,fk,fm,one=1.0_iwp,          &
pt5=0.5_iwp,theta,time,two=2.0_iwp,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
LOGICAL::consistent=.FALSE.

!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:),g_g(:,:,),g_num(:,:,),kdiag(:,nf(:,:,), &
node(:,num(:)
REAL(iwp),ALLOCATABLE::bee(:,coord(:,dee(:,der(:,deriv(:,&
d1x0(:,d1x1(:,d2x0(:,d2x1(:,ecm(:,fun(:,f1(:,g_coord(:,&
jac(:,km(:,kv(:,loads(:,mm(:,mv(:,points(:,prop(:,&
val(:,weights(:,x0(:,x1(:,x_coords(:,y_coords(:)

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),           &
dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),&
deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),num(nod),g_num(nod,nels),     &
g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),fun(nod),etype(nels),        &
prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtim,nstep,theta,npri,nres,fm,fk
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formmf(nf); neq=MAXVAL(nf)
ALLOCATE(x0(0:neq),d1x0(0:neq),x1(0:neq),d2x0(0:neq),loads(0:neq),          &
d1x1(0:neq),d2x1(0:neq),kdiag(neq))
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
CALL sample(element,points,weights); kdiag=0

!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g; CALL fkdiag(kdiag,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
DO i=2,neq; kdiag(i)=kdiag(i)+kdiag(i-1); END DO
ALLOCATE(kv(kdiag(neq)),mv(kdiag(neq)),f1(kdiag(neq)))
WRITE(11,'(2(A,I5))')                                     &
" There are",neq," equations and the skyline storage is",kdiag(neq)
kv=zero; mv=zero

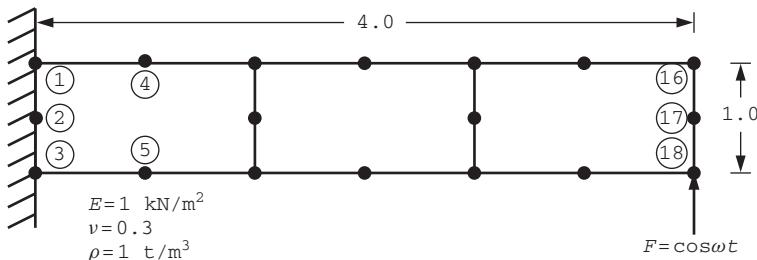
```

```

!-----global stiffness and mass matrix assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    g=g_g(:,iel); km=zeros; mm=zeros; area=zeros
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        area=area+det*weights(i)
        IF(consistent)THEN; CALL ecmat(ecm,fun,ndof,nodof)
            mm=mm+ecm*det*weights(i)*prop(3,etype(iel))
        END IF
    END DO gauss_pts_1
    IF(.NOT.consistent)CALL elmat(area,prop(3,etype(iel)),mm)
    CALL fsparv(kv,km,g,kdiag); CALL fsparv(mv,mm,g,kdiag)
END DO elements_2
!-----initial conditions and factorise equations-----
x0=zeros; d1x0=zeros; d2x0=zeros
c1=d6/(theta*dtim)**2; c2=c1*theta*dtim; c3=dtim**2/d6; c4=pt5*theta*dtim
f1=(c1+pt5*c2*fm)*mv+(one+pt5*c2*fk)*kv; CALL sparin(f1,kdiag); time=zero
!-----time stepping loop-----
WRITE(11,'(/A,I5)')" Result at node",nres
WRITE(11,'(A)')" time load x-disp y-disp"
WRITE(11,'(4E12.4)')time,load(time),x0(:,nres))
timesteps: DO j=1,nstep
    time=time+dtime; loads=zeros
    x1=(c1+pt5*c2*fm)*x0+(c2+two*fm)*d1x0+(two+c4*fm)*d2x0
    DO i=1,loaded_nodes
        loads(nf(:,node(i)))=
            val(i,:)*(theta*load(time)+(one-theta)*load(time-dtime)) &
        END DO; CALL linmul_sky(mv,x1,d1x1,kdiag); d1x1=loads+d1x1
    loads=pt5*c2*fk*x0+two*fk*d1x0+c4*fk*d2x0
    CALL linmul_sky(kv,loads,x1,kdiag); x1=x1+d1x1; CALL spabac(f1,x1,kdiag)
    d2x1=d2x0+(x1-x0)*c1-d1x0*c2-d2x0*two-d2x0)/theta
    d1x1=d1x0+pt5*dtim*(d2x1+d2x0); x0=x0+dtime*d1x0+two*c3*d2x0+d2x1*c3
    d1x0=d1x1; d2x0=d2x1
    IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,load(time),x0(:,nres))
END DO timesteps
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!-----Load-time function-----
IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
REAL(iwp),INTENT(IN)::t
REAL(iwp)::load_result
load_result=COS(0.3_iwp*t)
RETURN
END FUNCTION load
END PROGRAM p114

```

This algorithm is described in Section 3.13.3. The essential step is that shown in (3.148), which is of exactly the same form as (3.145) for the ‘theta’ method, and so this program can be expected to resemble the previous one very closely. The structure chart of Figure 11.8 is again appropriate and the problem layout and data given in Figure 11.11 are



```

nx e   ny e
3      1

np_types
1

prop(e,v,rho)
1.0  0.3  1.0

etype (not needed)

x_coords, y_coords
0.0  1.33333  2.66667  4.0
0.0  -1.0

dtim nstep theta npri nres fm fk
1.0  20    0.5   1     18   0.005  0.272

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

loaded_nodes,(node(i),val(i,:),i=1,loaded_nodes)
1
18 0.0 1.0

```

**Figure 11.11** Mesh and data for Program 11.4 example

There are 30 equations and the skyline storage is 285

```

Result at node 18
time      load      x-disp      y-disp
0.0000E+00  0.1000E+01  0.0000E+00  0.0000E+00
0.1000E+01  0.9553E+00  0.1544E+00  0.4329E+00
0.2000E+01  0.8253E+00  0.9763E+00  0.2575E+01
0.3000E+01  0.6216E+00  0.2083E+01  0.4966E+01
0.4000E+01  0.3624E+00  0.2908E+01  0.7004E+01
0.5000E+01  0.7074E-01  0.3396E+01  0.9110E+01
0.6000E+01  -0.2272E+00  0.3657E+01  0.1117E+02
0.7000E+01  -0.5048E+00  0.3581E+01  0.1253E+02
0.8000E+01  -0.7374E+00  0.3002E+01  0.1270E+02
0.9000E+01  -0.9041E+00  0.1916E+01  0.1157E+02
0.1000E+02  -0.9900E+00  0.4607E+00  0.9270E+01
0.1100E+02  -0.9875E+00  -0.1204E+01  0.5928E+01
0.1200E+02  -0.8968E+00  -0.2935E+01  0.1730E+01
0.1300E+02  -0.7259E+00  -0.4592E+01  -0.3044E+01
0.1400E+02  -0.4903E+00  -0.6033E+01  -0.8013E+01
0.1500E+02  -0.2108E+00  -0.7122E+01  -0.1275E+02
0.1600E+02  0.8750E-01  -0.7739E+01  -0.1682E+02
0.1700E+02  0.3780E+00  -0.7769E+01  -0.1984E+02
0.1800E+02  0.6347E+00  -0.7121E+01  -0.2149E+02
0.1900E+02  0.8347E+00  -0.5759E+01  -0.2156E+02
0.2000E+02  0.9602E+00  -0.3736E+01  -0.2002E+02

```

**Figure 11.12** Results from Program 11.4 example

essentially the same as in Figure 11.9. No new variables are involved, but the parameter  $\theta$  is set to its stability limit of about 1.4, compared with 0.5 in the previous algorithm. All that need be said is that the  $f_1$  matrix is now constructed as demanded by (3.148). The remaining steps of (3.150)–(3.153) are carried out within the section headed ‘time-stepping loop’.

The results for one cycle are listed as Figure 11.12 and plotted in Figure 11.7. Again, the early stages of the response reflect mainly the influence of the start-up conditions. The response eventually settles down to be in good agreement with the previous solutions, although with a slightly greater phase shift.

## Program 11.5 Forced vibration of a rectangular elastic solid in plane strain using 8-node quadrilateral elements numbered in the y-direction. Lumped mass, complex response

```

PROGRAM p115
!-----
! Program 11.5 Forced vibration of a rectangular elastic solid in plane
! strain using uniform 8-node quadrilaterals. Lumped mass.
! Mesh numbered in y-direction. Complex response.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,iel,k,nband,ndim=2,ndof=16,nels,neq,nip=9,nlen,nn,nod=8,
         nodof=2,npri,nprops=4,np_types,nr,nres,nst=3,nstep,nxe,nye
REAL(iwp)::a,area,b,det,dr,dtim,d2=2.0_iwp,omega=0.3,one=1.0_iwp,
           & time,zero=0.0_iwp
CHARACTER(len=15)::argv,element='quadrilateral'
!----- dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:,),g_num(:,:,),nf(:,:,),num(:)
REAL(iwp),ALLOCATABLE::bee(:,cm(:,coord(:,dee(:,der(:,&
         deriv(:,g_coord(:,jac(:,km(:,mm(:,points(:,prop(:,&
         weights(:,x_coords(:,y_coords(:,COMPLEX(iwp),ALLOCATABLE::kc(:),loads(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof), g_coord(ndim,nn),
         & dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),
         & deriv(ndim,nod),bee(nst,ndof),km(ndof,ndof),num(nod),g_num(nod,nels),
         & g_g(ndof,nels),mm(ndof,ndof),cm(ndof,ndof),prop(nprops,np_types),
         & etype(nels),x_coords(nxe+1),y_coords(nye+1))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtim,nstep,npri,nres
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
CALL sample(element,points,weights); nband=0
!-----loop the elements to find bandwidth-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    call num_to_g(num,nf,g); g_num(:,iel)=num; g_g(:,iel)=g
    g_coord(:,num)=transpose(coord)
    IF(nband<bandwidth(g))nband=bandwidth(g)
END DO elements_1

```

```

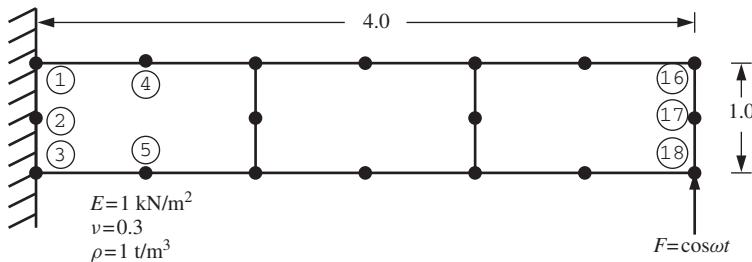
WRITE(11, '(2(A,I5))')
&
" There are ",neq," equations and the half-bandwidth is", nband
ALLOCATE(kc(neq*(nband+1)), loads(0:neq)); kc=(0.0_iwp,0.0_iwp)
!----- element stiffness integration and assembly-----
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); coord=transpose(g_coord(:,num))
    g=g_g(:,iel); area=(coord(5,1)-coord(1,1))*(coord(5,2)-coord(1,2))
    CALL elmat(area,prop(3,etype(iel)),mm); km=zero
    gauss_points_1: DO i=1,nip; CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_points_1
    dr=prop(4,etype(iel)); cm=km*d2*dr*SQRT(one-dr*dr)
    km=km*(one-d2*dr*dr)-mm*omega**2; CALL formkc(kc,km,cm,g,neq)
END DO elements_2
!-----complex equation solution-----
loads=(0.0_iwp,0.0_iwp); loads(neq)=(1.0_iwp,0.0_iwp)
CALL comred(kc,neq); CALL comsub(kc,loads)
a=REAL(loads(neq)); b=AIMAG(loads(neq))
!----- time stepping loop -----
WRITE(11, '(/A,I5)')" Result at node",nres
WRITE(11,'(A)')"      time          load          y-disp"
timesteps: DO i=0,nstep
    time=i*dtim; IF(i/npri*npri==i)WRITE(11,'(3E12.4)')time,
    & COS(omega*time),a*COS(omega*time)-b*SIN(omega*time)
    END DO timesteps
STOP
END PROGRAM p115

```

This forced vibration analysis uses the complex response method described in Section 3.13.4. The program terminology is essentially as before, but as shown in (3.157) and (3.158), the equations to be solved involve a left-hand-side coefficient matrix and right-hand-side vector that both contain complex numbers. Thus the appropriate assembly routine is `formkc` and the appropriate equation solution routines are `comred` and `combac`. The same example as tackled previously using Programs 11.2, 11.3 and 11.4 is considered once more, as shown in Figure 11.13. From the data, it can be seen that the damping ratio  $\gamma$  is treated as a material property, so each element has four properties, namely  $E$ ,  $\nu$ ,  $\rho$  and  $\gamma$ .

The element stiffness matrix is integrated in the usual way and an element lumped mass matrix is generated by routine `elmat`. Prior to assembly, as required by (3.157) and (3.158), the real and imaginary parts of the element complex stiffness matrix are created. The imaginary part called `cm` is created from `km` multiplied by the factor  $2\gamma\sqrt{1-\gamma^2}$ . The real part, still called `km`, incorporates the factor  $(1-2\gamma^2)$  and is subtracted by the mass matrix multiplied by the factor  $\omega^2$ .

The results shown in Figure 11.14 indicate virtually the same amplitude and phase shift as computed by the modal superposition Program 11.2. The result over 100 time steps is compared with the other methods in Figure 11.7.



```

nx e   ny e
3      1

np_types
1

prop(e,v,rho,dr)
1.0  0.3  1.0  0.05

etype (not needed)

x_coords, y_coords
0.0 1.33333  2.66667  4.0
0.0  -1.0

dtim nstep npri nres
1.0  20    1     18

nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0

```

**Figure 11.13** Mesh and data for Program 11.5 example

There are 30 equations and the half-bandwidth is 15

```

Result at node 18
      time      load      y-disp
0.0000E+00  0.1000E+01  0.1509E+02
0.1000E+01  0.9553E+00  0.2672E+02
0.2000E+01  0.8253E+00  0.3596E+02
0.3000E+01  0.6216E+00  0.4200E+02
0.4000E+01  0.3624E+00  0.4428E+02
0.5000E+01  0.7074E-01  0.4260E+02
0.6000E+01  -0.2272E+00 0.3712E+02
0.7000E+01  -0.5048E+00 0.2833E+02
0.8000E+01  -0.7374E+00 0.1700E+02
0.9000E+01  -0.9041E+00 0.4157E+01
0.1000E+02  -0.9900E+00 -0.9059E+01
0.1100E+02  -0.9875E+00 -0.2147E+02
0.1200E+02  -0.8968E+00 -0.3195E+02
0.1300E+02  -0.7259E+00 -0.3959E+02
0.1400E+02  -0.4903E+00 -0.4369E+02
0.1500E+02  -0.2108E+00 -0.4388E+02
0.1600E+02  0.8750E-01 -0.4016E+02
0.1700E+02  0.3780E+00 -0.3285E+02
0.1800E+02  0.6347E+00 -0.2260E+02
0.1900E+02  0.8347E+00 -0.1034E+02
0.2000E+02  0.9602E+00 0.2850E+01

```

**Figure 11.14** Results from Program 11.5 example

## 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

```

PROGRAM p116
!-----
! Program 11.6 Forced vibration analysis of an elastic solid in plane
! strain using rectangular uniform size 4-node quadrilaterals.
! Mesh numbered in the x- or y-direction. Lumped and/or
! consistent mass. Mixed explicit/implicit time integration.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,j,iel,k,ndim=2,ndof=8,nels,neg,nip=4,nlen,nn,nodof=4, nodof=2,      &
npri,nprops=3,np_types,nr,nres,nstep,nxe,nye
REAL(iwp)::area,beta,c1,det,dtim,gamma,one=1.0_iwp,pt5=0.5_iwp,time,          &
two=2.0_iwp,zero=0.0_iwp; CHARACTER(LEN=15)::argv;element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:, :,g_num(:, :,kdiag_l(:,      &
kdiag_r(:,nf(:, :,num(:,      &
REAL(iwp),ALLOCATABLE::coord(:, :,der(:, :,d1x0(:, d1x1(:, d2x0(:,      &
d2x1(:, ecm(:, :,fun(:, g_coord(:, :,jac(:, :,km(:, :,kv(:, mm(:, :,      &
mv(:, points(:, :,prop(:, :,weights(:, x0(:, x1(:, x_coords(:,      &
y_coords(:, CHARACTER(LEN=1),ALLOCATABLE::mtype(:,      &
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn),      &
coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod),      &
km(ndof,ndof),num(nod),g_num(nod,nels),g_g(ndof,nels),mtype(nels),      &
mm(ndof,ndof),ecm(ndof,ndof),fun(nod),prop(np	props,np_types),      &
x_coords(nxe+1),y_coords(nye+1),etype(nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtim,nstep,beta,gamma,npri,nres
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(kdiag_l(neq),kdiag_r(neq),x0(0:neq),d1x0(0:neq),x1(0:neq),      &
d2x0(0:neq),d1x1(0:neq),d2x1(0:neq))
READ(10,*)mtype; CALL sample(element,points,weights); kdiag_l=0; kdiag_r=0
!-----loop the elements to find global array sizes-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
    CALL fkdiag(kdiag_r,g); IF(mtype(iel)=='c')CALL fkdiag(kdiag_l,g)
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
WHERE(kdiag_l==0); kdiag_l=1; END WHERE
DO i=2,neq; kdiag_l(i)=kdiag_l(i)+kdiag_l(i-1)
    kdiag_r(i)=kdiag_r(i)+kdiag_r(i-1)
END DO; ALLOCATE(kv(kdiag_l(neq)),mv(kdiag_r(neq)))
WRITE(11,'(A,I5,A,/,2(A,I5),A)')
    " There are",neq," equations.",," Skyline storage is",kdiag_l(neq),      &
    " to the left, and",kdiag_r(neq)," to the right."
    cl=one/dtim/dtim/beta; kv=zero; mv=zero

```

```

!-----global stiffness and mass matrix assembly-----
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num))
    CALL stiff4(km,coord,prop(1,etype(iel)),prop(2,etype(iel)))
    g=g_g(:,iel); area=zero; mm=zero
    gauss_pts_1: DO i=1,nip; CALL shape_der(der,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); area=area+det*weights(i)
        CALL shape_fun(fun,points,i)
        IF(mtype(iel)=='c')THEN; CALL ecmat(ecm,fun,ndof,nodof)
            mm=mm+ecm*det*weights(i)*c1*prop(3,etype(iel))
        END IF
    END DO gauss_pts_1
    IF(mtype(iel)=='l')THEN
        CALL elmat(area,c1*prop(3,etype(iel)),mm)
        CALL fsparv(kv,mm,g,kdiag_l); CALL fsparv(mv,mm-km,g,kdiag_r); ELSE
        CALL fsparv(kv,km+mm,g,kdiag_l); CALL fsparv(mv,mm,g,kdiag_r)
    END IF
END DO elements_2
!-----initial conditions and factorise equations-----
x0=zero; d1x0=one; d2x0=zero; CALL sparin(kv,kdiag_l); time=zero
!-----time stepping loop-----
WRITE(11,'(/A,I5)')" Result at node",nres
WRITE(11,'(A)')" time x-disp y-disp"
WRITE(11,'(4E12.4)')time,x0(nf(:,nres))
timesteps: DO j=1,nstep
    time=time+dtime; d1x1=x0+d1x0*dtime+d2x0*pt5*dtim*dtim*(one-two*beta)
    CALL linmul_sky(mv,d1x1,x1,kdiag_r); CALL spabac(kv,x1,kdiag_l)
    d2x1=(x1-d1x1)/dtime/dtime/beta
    d1x1=d1x0+d2x0*dtim*(one-gamma)+d2x1*dtim*gamma
    x0=x1; d1x0=d1x1; d2x0=d2x1
    IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,x0(nf(:,nres))
END DO timesteps
STOP
END PROGRAM p116

```

Programs 11.3 and 11.4 used implicit time-marching algorithms, and Program 11.8 described later in this chapter uses an explicit approach. The program described now combines the methods of implicit and explicit time integration in a single program. Although not ‘element-by-element’, this procedure should allow economical bandwidths of the assembled matrices. The idea (e.g., Key, 1980) is that a mesh may contain only a few elements which have a very small explicit stability limit and are therefore best integrated implicitly. The remainder of the mesh can be successfully integrated explicitly at reasonable time steps.

The recurrence relations (3.145)–(3.147) are cast in the form

$$\left( \frac{1}{\Delta t^2 \beta} [\mathbf{M}_m] + [\mathbf{K}_m] \right) \{\mathbf{U}\}_1 = \{\mathbf{F}\}_1 + \frac{1}{\Delta t^2 \beta} [\mathbf{M}_m] \{\bar{\mathbf{U}}\}_1 \quad (11.4)$$

for implicit elements, and

$$\frac{1}{\Delta t^2 \beta} [\mathbf{M}_m] \{\mathbf{U}\}_1 = \{\mathbf{F}\}_1 + \left( \frac{1}{\Delta t^2 \beta} [\mathbf{M}_m] - [\mathbf{K}_m] \right) \{\bar{\mathbf{U}}\}_1 \quad (11.5)$$

for explicit elements, where

$$\{\bar{\mathbf{U}}\}_1 = \{\mathbf{U}\}_0 + \Delta t \{\dot{\mathbf{U}}\}_0 + \frac{\Delta t^2 (1 - 2\beta)}{2} \{\ddot{\mathbf{U}}\}_0 \quad (11.6)$$

Accelerations and velocities are obtained from

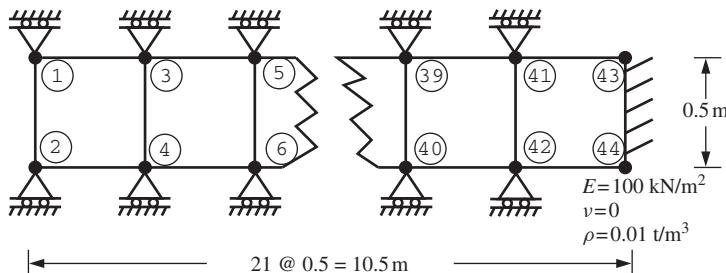
$$\{\ddot{\mathbf{U}}\}_1 = \frac{1}{\Delta t^2 \beta} (\{\mathbf{U}\}_1 - \{\bar{\mathbf{U}}\}_1) \quad (11.7)$$

and

$$\{\dot{\mathbf{U}}\}_1 = \{\dot{\mathbf{U}}\}_0 + \Delta t(1 - \gamma)\{\ddot{\mathbf{U}}\}_0 + \Delta t\gamma\{\ddot{\mathbf{U}}\}_1 \quad (11.8)$$

The time-integration parameters are the ‘Newmark’ ones introduced in Program 11.1 and conventionally set to  $\beta = 1/4$  and  $\gamma = 1/2$ , corresponding to  $\theta = 1/2$  as used in Program 11.3.

When an explicit element is not coupled to an implicit one, the half-bandwidth (excluding the diagonal) of the assembled equation coefficient matrix will only be 1, whereas the full half-bandwidth will apply for implicit elements. Full advantage can be taken of variable bandwidth or ‘skyline’ storage on both sides of the equation in this case (e.g., Smith, 1984). The problem chosen is illustrated in Figure 11.15 and models the impact of an elastic rod, initially travelling at a uniform unit velocity, with a rigid wall. The elastic



```

nxe  nye  np_types
21    1    1

prop(e,v,rho)
100.0  0.0  0.01

etype (not needed)

x_coords, y_coords
0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0
5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5  10.0 10.5
0.0 -0.5

dtim   nstep  beta  gamma  npri  nres
0.0025  400    0.25   0.5     1      42

nr, (k,nf(:,k),i=1,nr)
44
  1 1 0  2 1 0  3 1 0  4 1 0  5 1 0
  6 1 0  7 1 0  8 1 0  9 1 0 10 1 0
11 1 0 12 1 0 13 1 0 14 1 0 15 1 0
16 1 0 17 1 0 18 1 0 19 1 0 20 1 0
21 1 0 22 1 0 23 1 0 24 1 0 25 1 0
26 1 0 27 1 0 28 1 0 29 1 0 30 1 0
31 1 0 32 1 0 33 1 0 34 1 0 35 1 0
36 1 0 37 1 0 38 1 0 39 1 0 40 1 0
41 1 0 42 1 0 43 0 0 44 0 0

mtype
'c' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' 'c'
'c' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' 'c'

```

**Figure 11.15** Mesh and data for Program 11.6 example

rod is constrained to vibrate in the axial direction only, and is fixed at the right-hand end. Initial conditions of a uniform unit velocity are applied to all freedoms in the mesh at time  $t = 0$ . The appropriate structure chart is Figure 11.8 for implicit integration.

No damping is considered in this case, and the data calls for nstep=400 calculation steps at a time step of dtim=0.0025. Output is requested every npri=1 time steps at node nres=42, which is close to the impacted end of the rod. The boundary conditions place rollers on the top and bottom surfaces of the rod, although there would be no tendency for ‘bulging’ in this case since Poisson’s ratio is read as zero.

A new dynamic character array mtype is introduced, which is assigned from data, and holds for each element, the character string ‘c’ or ‘l’, corresponding to consistent or lumped mass, respectively. In the present example, elements 1, 11 and 21 have consistent mass while the others are lumped. Also, the lumped elements are explicitly integrated while the consistent ones are implicitly integrated.

The stiffness and mass are integrated as usual. When the element mass matrix is consistent ( $mtype='c'$ ),  $mm+km$  is assembled into  $kv$  while the  $mm$  is assembled into  $mv$  as shown in (11.4) for implicit elements. Conversely, when the element mass matrix is lumped ( $mtype='l'$ ),  $mm$  is assembled into  $kv$  while  $(mm-km)$  is assembled into  $mv$  as shown in (11.5) for explicit elements. The integer vectors kdiag\_l and kdiag\_r locate the diagonal terms of the left- and right-hand-side matrices  $kv$  and  $mv$ , respectively.

The initial conditions are then set, with the starting velocity in the  $x$ -direction at all nodes set to unity ( $d1x0=one$ ). The global matrix factorisation is done by sparin and the time-stepping loop is entered. Equations (11.4) and (11.5) require the usual matrix-by-vector multiplication on the right-hand side by subroutine linmul\_sky. Equation solution is completed by spabac and it remains only to update accelerations and velocities for the next time step from (11.7) and (11.8).

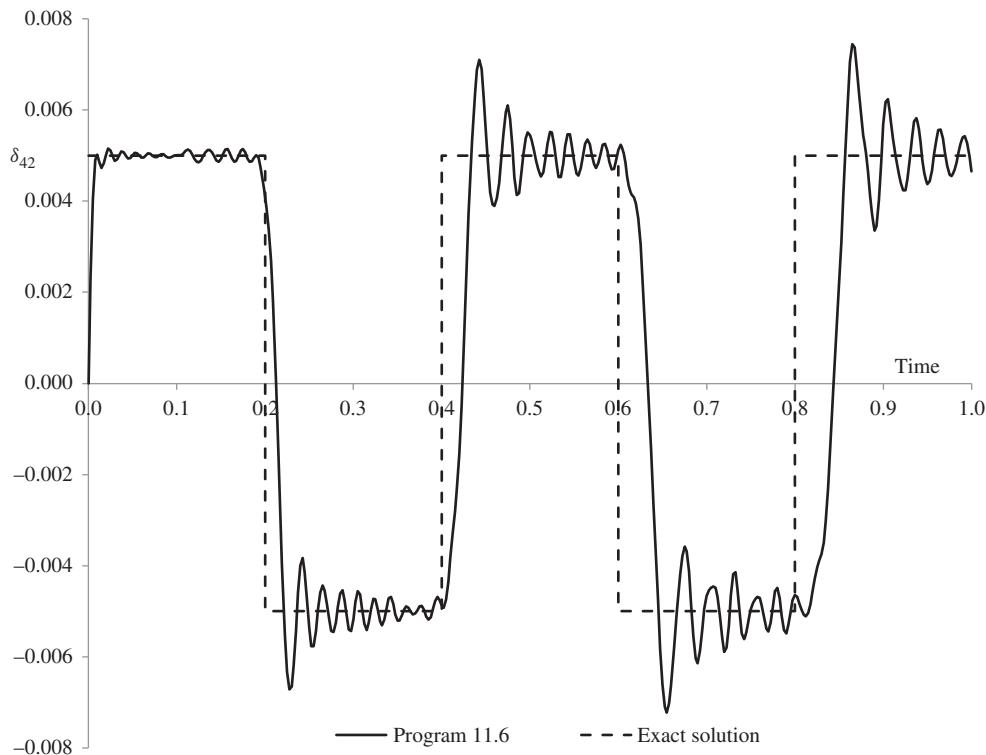
The results are listed in Figure 11.16 and the displacements close to the support (freedom 42) are compared graphically with the exact solution in Figure 11.17. Despite some spurious oscillations, the response is reasonably modelled.

```
There are 42 equations.
Skyline storage is 55 to the left, and 143 to the right.

Result at node 42
  time      x-disp      y-disp
  0.0000E+00  0.0000E+00  0.0000E+00
  0.2500E-02  0.2326E-02  0.0000E+00
  0.5000E-02  0.4051E-02  0.0000E+00
  0.7500E-02  0.4898E-02  0.0000E+00
  0.1000E-01  0.5020E-02  0.0000E+00
  0.1250E-01  0.4834E-02  0.0000E+00
  0.1500E-01  0.4724E-02  0.0000E+00
  0.1750E-01  0.4825E-02  0.0000E+00
  0.2000E-01  0.5029E-02  0.0000E+00
  0.2250E-01  0.5150E-02  0.0000E+00
  0.2500E-01  0.5106E-02  0.0000E+00

  .
  .
  .
  0.9775E+00  0.4544E-02  0.0000E+00
  0.9800E+00  0.4640E-02  0.0000E+00
  0.9825E+00  0.4821E-02  0.0000E+00
  0.9850E+00  0.5040E-02  0.0000E+00
  0.9875E+00  0.5250E-02  0.0000E+00
  0.9900E+00  0.5399E-02  0.0000E+00
  0.9925E+00  0.5420E-02  0.0000E+00
  0.9950E+00  0.5270E-02  0.0000E+00
  0.9975E+00  0.4973E-02  0.0000E+00
  0.1000E+01  0.4648E-02  0.0000E+00
```

**Figure 11.16** Results from Program 11.6 example



**Figure 11.17** Displacement at node 42 vs. time from Program 11.6 example

**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

```

PROGRAM p117
!-----
!  
! 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 x- or y-direction.  
! Implicit time integration using the "theta" method.  
! No global matrix assembly. Diagonally preconditioned  
! conjugate gradient solver.
!  
!
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::cg_iters,cg_limit,i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels, &
neq,nip=9,nlen,nn,nod=8,nodof=2,npri,nprops=3,np_types,nr,nres,nst=3, &
nstep,nxe,nye
REAL(iwp)::alpha,area,beta,cg_tol,c1,c2,c3,c4,det,dtim,fk,fm,one=1.0_iwp,&
theta,time,up,zero=0.0_iwp; LOGICAL::consistent=.FALSE.,cg_converged
CHARACTER(LEN=15)::argv,element='quadrilateral'

```

```

!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:, :,),g_num(:, :,),nf(:, :,),node(:, ), &
num(:)

REAL(iwp),ALLOCATABLE::bee(:, :,coord(:, :,d(:, ),dee(:, :,der(:, :, &
deriv(:, :,diag_precon(:, d1x0(:, d1x1(:, d2x0(:, d2x1(:, ecm(:, :, &
fun(:, g_coord(:, :,jac(:, :,km(:, :,loads(:, mm(:, :,p(:, points(:, :,) &
prop(:, :,storkm(:, :,stomm(:, :,u(:, val(:, :,weights(:, x(:, &
xnew(:, x0(:, x1(:, x_coords(:, y_coords(:, &

!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye)
ALLOCATE(nf(nodof,nn),points(nip,ndim),g(ndof),g_coord(ndim,nn), &
dee(nst,nst),coord(nod,ndim),jac(ndim,ndim),weights(nip),der(ndim,nod), &
deriv(ndim,nod),bee(nst,nodof),km(ndof,ndof),num(nod),g_num(nod,nels), &
g_g(ndof,nels),mm(ndof,ndof),ecm(ndof,ndof),fun(nod),etype(nels), &
prop(nprops,np_types),x_coords(nxe+1),y_coords(nye+1), &
storkm(ndof,ndof,nels),stomm(ndof,ndof,nels))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtim,nstep,theta,npri,nres,fm,fk
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(x0(0:neq),d1x0(0:neq),x1(0:neq),d2x0(0:neq),loads(0:neq), &
d1x1(0:neq),d2x1(0:neq),d(0:neq),p(0:neq),x(0:neq),xnew(0:neq), &
diag_precon(0:neq),u(0:neq))
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
!-----loop the elements to set up element data-----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1
CALL sample(element,points,weights); diag_precon=zero
WRITE(11,'(A,I5,A)')" There are",neq," equations"
c1=(one-theta)*dtim; c2=fk-c1; c3=fm+one/(theta*dtim); c4=fk+theta*dtim
CALL sample(element,points,weights); diag_precon=zero
!----element stiffness and mass integration, storage and preconditioner---
elements_2: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    km=zero; mm=zero; area=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        jac=MATMUL(der,coord); det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
        area=area+det*weights(i)
        IF(consistent)THEN; CALL ecmat(ecm,fun,ndof,nodof)
            ecm=ecm*det*weights(i); mm=mm+ecm
        END IF
    END DO gauss_pts_1
    IF(.NOT.consistent)CALL elmat(area,prop(3,etype(iel)),mm)
    storkm(:, :,iel)=km; stomm(:, :,iel)=mm
    DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+mm(k,k)*c3+km(k,k)*c4
    END DO
END DO elements_2
diag_precon(1:)=one/diag_precon(1:); diag_precon(0)=zero

```

```

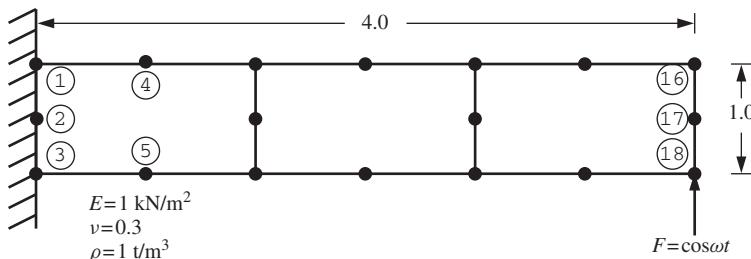
!-----time stepping loop-----
x0=zero; d1x0=zero; d2x0=zero; time=zero
WRITE(11,'(/A,I5)')" Result at node",nres
WRITE(11,'(A)')
      "      time      load      x-disp      y-disp      cg iters"
      &
WRITE(11,'(4E12.4)')time,load(time),x0(nf(:,nres))
timesteps: DO j=1,nstep
    time=time+dtim; loads=zero; u=zero
    elements_3: DO iel=1,nel
        g=g_g(:,iel); km=storkm(:,:,iel); mm=stormmm(:,:,iel)
        u(g)=u(g)+MATMUL(km*c2+mm*c3,x0(g))+MATMUL(mm/theta,d1x0(g))
    END DO elements_3; u(0)=zero
    DO i=1,loaded_nodes
        loads(nf(:,node(i)))=
            val(i,:)*(theta*dtim*load(time)+c1*load(time-dtim))
    END DO; loads=u+loads; d=diag_precon*loads; p=d; x=zero; cg_iters=0
!-----pcg equation solution-----
pcg: DO
    cg_iters=cg_iters+1; u=zero
    elements_4: DO iel=1,nel
        g=g_g(:,iel); km=storkm(:,:,iel); mm=stormmm(:,:,iel)
        u(g)=u(g)+MATMUL(mm*c3+km*c4,p(g))
    END DO elements_4; u(0)=zero
    up=DOT_PRODUCT(loads,d); alpha=up/DOT_PRODUCT(p,u); xnew=x+p*alpha
    loads=loads-u*alpha; d=diag_precon*loads; beta=DOT_PRODUCT(loads,d)/up
    p=d+p*beta; call checon(xnew,x,cg_tol,cg_converged)
    IF(cg_converged.OR.cg_iters==cg_limit)EXIT
END DO pcg
x1=xnew; d1x1=(x1-x0)/(theta*dtim)-d1x0*(one-theta)/theta
d2x1=(d1x1-d1x0)/(theta*dtim)-d2x0*(one-theta)/theta
IF(j/npri*npri==j)
    WRITE(11,'(4E12.4,I8)')time,load(time),x1(nf(:,nres)),cg_iters
    x0=x1; d1x0=d1x1; d2x0=d2x1
    &
END DO timesteps
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!-----Load-time function-----
IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
REAL(iwp),INTENT(IN)::t
REAL(iwp)::load_result
load_result=COS(0.3_iwp*t)
RETURN
END FUNCTION load
END PROGRAM p117

```

‘Element-by-element’ strategies can preserve the unconditional stability of the ‘implicit’ procedures such as in Program 11.3 by replacing the direct equation solution by an iterative approach such as pcg. Alternatively, as was done in Chapter 8 (Program 8.6), a purely explicit time-integration procedure for dynamic analysis can be adopted with its inherent stability limitations, as will be explained in the final Program 11.8. Product EBE techniques are also available (Wong *et al.*, 1989) but are not dealt with in this book.

Program 11.7 is adapted from Program 11.3 (implicit integration by the ‘theta’ method) in the same way as Program 8.5 was adapted from Program 8.4. Equation solution is accomplished on every time step by preconditioned conjugate gradients using diagonal preconditioning of the left-hand-side matrix in equation (3.145).

The element stiffness matrices  $\mathbf{km}$  are stored as  $\text{storkm}$  with the mass matrices  $\mathbf{mm}$  as  $\text{stomm}$ . The same example analysed by Programs 11.2, 11.3, 11.4 and 11.5 is repeated, with the data given in Figure 11.18. The results shown in Figure 11.19 are essentially the same as those obtained using a direct solver shown in Figure 11.10. It may also be noted from these results that this version of pcg solution is taking approximately  $\text{neq}/2$  iterations to converge (where  $\text{neq}$  is the number of equations), a similar convergence rate as was usual in Chapters 5 and 6. In Chapter 8, the convergence rate was much quicker at approximately  $\text{neq}/10$ . Fortunately, as problem sizes increase, the iteration count as a proportion of  $\text{neq}$ , drops very rapidly.



```

nx e   ny e   cg tol   cg limit
3      1     1.0e-5    50
np types
1
prop(e,v,rho)
1.0  0.3  1.0
etype (not needed)
x coords, y coords
0.0  1.33333  2.66667  4.0
0.0  -1.0
dtim  nstep  theta  npri  nres  fm    fk
1.0    20     0.5    1     18   0.005  0.272
nr,(k,nf(:,k),i=1,nr)
3
1 0 0  2 0 0  3 0 0
loaded nodes, (node(i),val(i,:),i=1,loaded nodes)
1
18 0.0 1.0

```

**Figure 11.18** Mesh and data for Program 11.7 example

There are 30 equations

Result at node 18	time	load	x-disp	y-disp	cg	iters
0.0000E+00	0.1000E+01	0.0000E+00	0.0000E+00			
0.1000E+01	0.9553E+00	0.7363E+00	0.2167E+01		17	
0.2000E+01	0.8253E+00	0.2231E+01	0.5226E+01		17	
0.3000E+01	0.6216E+00	0.3252E+01	0.7398E+01		17	
0.4000E+01	0.3624E+00	0.3987E+01	0.1046E+02		17	
0.5000E+01	0.7074E-01	0.4832E+01	0.1398E+02		16	
0.6000E+01	-0.2272E+00	0.5272E+01	0.1678E+02		16	
0.7000E+01	-0.5048E+00	0.5086E+01	0.1839E+02		16	
0.8000E+01	-0.7374E+00	0.4370E+01	0.1894E+02		17	
0.9000E+01	-0.9041E+00	0.3255E+01	0.1834E+02		17	
0.1000E+02	-0.9900E+00	0.1789E+01	0.1650E+02		17	
0.1100E+02	-0.9875E+00	0.8842E-01	0.1349E+02		18	
0.1200E+02	-0.8968E+00	-0.1674E+01	0.9574E+01		18	
0.1300E+02	-0.7259E+00	-0.3373E+01	0.5082E+01		18	
0.1400E+02	-0.4903E+00	-0.4916E+01	0.3731E+00		17	
0.1500E+02	-0.2108E+00	-0.6151E+01	-0.4127E+01		17	
0.1600E+02	0.8750E-01	-0.6859E+01	-0.7907E+01		17	
0.1700E+02	0.3780E+00	-0.6839E+01	-0.1050E+02		17	
0.1800E+02	0.6347E+00	-0.5986E+01	-0.1157E+02		18	
0.1900E+02	0.8347E+00	-0.4304E+01	-0.1096E+02		18	
0.2000E+02	0.9602E+00	-0.1902E+01	-0.8706E+01		18	

Figure 11.19 Results from Program 11.7 example

### 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

```

PROGRAM p118
!-----
! 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 x- or y-direction.
! Explicit time integration.
!-----
USE main; USE geom; IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
INTEGER::i,j,iel,k,loaded_nodes,ndim=2,ndof=16,nels,neq,nip=4,nlen,nn, &
nod=8,nodof=2,npri,nprops=4,np_types,nr,nres,nst=4,nstep,nxe,nye
REAL(iwp)::area,det,dsbar,dtim,f,fac,fmax,fnew,lode_theta,one=1.0_iwp, &
pt5=0.5_iwp,sigm,time,zero=0.0_iwp
CHARACTER(LEN=15)::argv,element='quadrilateral'
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:,g(:,g_g(:,:)),g_num(:,:,),nf(:,:,),node(:, ), &
num(:)
REAL(iwp),ALLOCATABLE::bdylds(:,bee(:,:,bload(:,coord(:,:,dee(:,:, ), &
der(:,:,),deriv(:,:,d1x1(:,d2x1(:,eld(:,eload(:,eps(:),
etensor(:,:,diag(:,g_coord(:,:,jac(:,:,mm(:,:,pl(:,:, ), &
points(:,:,prop(:,:,sigma(:,stress(:,tensor(:,:,val(:,:, ), &
weights(:,x1(:,x_coords(:,y_coords(:)
!-----input and initialisation-----
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nxe,nye,np_types; CALL mesh_size(element,nod,nels,nn,nxe,nye)

```

```

ALLOCATE(nf(nodof,nn),points(nip,ndim),weights(nip),g_coord(ndim,nn),      &
num(nod),dee(nst,nst),tensor(nst,nip,nels),coord(nod,ndim),pl(nst,nst),&
etensor(nst,nip,nels),jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),      &
g_num(nod,nels),bee(nst,ndof),eld(ndof),eps(nst),sigma(nst),      &
mm(ndof,ndof),bload(ndof),eload(ndof),g(ndof),stress(nst),etype(nels), &
g_g(ndof,nels),x_coords(nxe+1),y_coords(nye+1),prop(nprops,np_types))
READ(10,*)prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,dtim,nstep,npri,nres
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formnf(nf); neq=MAXVAL(nf)
ALLOCATE(bdylds(0:neq),x1(0:neq),d1x1(0:neq),d2x1(0:neq),diag(0:neq))
READ(10,*)loaded_nodes; ALLOCATE(node(loaded_nodes),val(loaded_nodes,ndim))
READ(10,*)(node(i),val(i,:),i=1,loaded_nodes)
!-----loop the elements to set up global geometry -----
elements_1: DO iel=1,nels
    CALL geom_rect(element,iel,x_coords,y_coords,coord,num,'Y')
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
END DO elements_1; CALL mesh(g_coord,g_num,argv,nlen,12)
WRITE(11,'(A,I5,A)')"There are",neq," equations"
!-----initial conditions-----
tensor=zero; etensor=zero; x1=zero; d1x1=zero; d2x1=zero; diag=zero
CALL sample(element,points,weights); time=zero
!-----time stepping loop-----
WRITE(11,'(/A,I5)')" Result at node",nres
WRITE(11,'(A)')"      time          load          x-disp          y-disp"
WRITE(11,'(4E12.4)')time,load(time),x1(nf(:,nres))
time_steps: DO j=1,nstep
    fmax=zero; time=time+dtim; x1=x1+dtim*d1x1+pt5*dtim**2*d2x1; bdylds=zero
!-----go round the Gauss Points -----
elements_2: DO iel=1,nels
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    area=zero; bload=zero; eld=x1(g)
    gauss_pts_1: DO i=1,nip; CALL shape_der(der,points,i)
        CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
        jac=MATMUL(der,coord); det=determinant(jac)
        area=area+det*weights(i); CALL invert(jac); deriv=MATMUL(jac,der)
        CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
        eps=eps-etensor(:,i,iel); sigma=MATMUL(dee,eps)
        stress=sigma+tensor(:,i,iel)
        CALL invar(stress,sigm,dsbar,lode_theta)
        fnew=dsbar-prop(4,etype(iel))
    gauss_pts_1: ENDDO
    !-----check whether yield is violated-----
    IF(fnew>=zero)THEN
        stress=tensor(:,i,iel); CALL invar(stress,sigm,dsbar,lode_theta)
        f=dsbar-prop(4,etype(iel)); fac=fnew/(fnew-f)
        stress=tensor(:,i,iel)+(one-fac)*sigma; CALL vmdpl(dee,stress,pl)
        dee=dee-fac*pl
    END IF
    sigma=MATMUL(dee,eps); sigma=sigma+tensor(:,i,iel)
    CALL invar(sigma,sigm,dsbar,lode_theta); f=dsbar-prop(4,etype(iel))
    IF(f>fmax)fmax=f; eload=MATMUL(sigma,bee)
    bload=bload+eload*det*weights(i)
!-----update the Gauss Point stresses and strains-----
    tensor(:,i,iel)=sigma; etensor(:,i,iel)=etensor(:,i,iel)+eps
END DO gauss_pts_1
bdylds(g)=bdylds(g)-bload
IF(j==1)THEN
    CALL elmat(area,prop(3,etype(iel)),mm); CALL formlump(diag,mm,g)

```

```

    END IF
END DO elements_2; bdylds(0)=zero
DO i=1,loaded_nodes
    bdylds(nf(:,node(i)))=bdylds(nf(:,node(i)))+val(i,:)*load(time)
END DO
bdylds(1:)=bdylds(1:)/diag(1:); d1x1=d1x1+(d2x1+bdylds)*pt5*dtim
d2x1=bdylds; WRITE(*,'(A,I6,A,F8.4)')" time step",j,"      F_max",fmax
IF(j/npri*npri==j)WRITE(11,'(4E12.4)')time,load(time),x1(nf(:,nres))
END DO time_steps
STOP
CONTAINS
FUNCTION load(t) RESULT(load_result)
!-----Load-time function-----
IMPLICIT NONE
INTEGER,PARAMETER::iwp=SELECTED_REAL_KIND(15)
REAL(iwp),INTENT(IN)::t
REAL(iwp)::load_result
load_result=-180.0_iwp
RETURN
END FUNCTION load
END PROGRAM p118

```

In the same way as was done for first-order problems in Program 8.6,  $\theta$  can be set to zero in second-order recurrence formulae such as (3.144). After rearrangement, the only matrix remaining on the left-hand side of the equation is  $[\mathbf{M}_m]$ . If this is lumped (diagonalised),

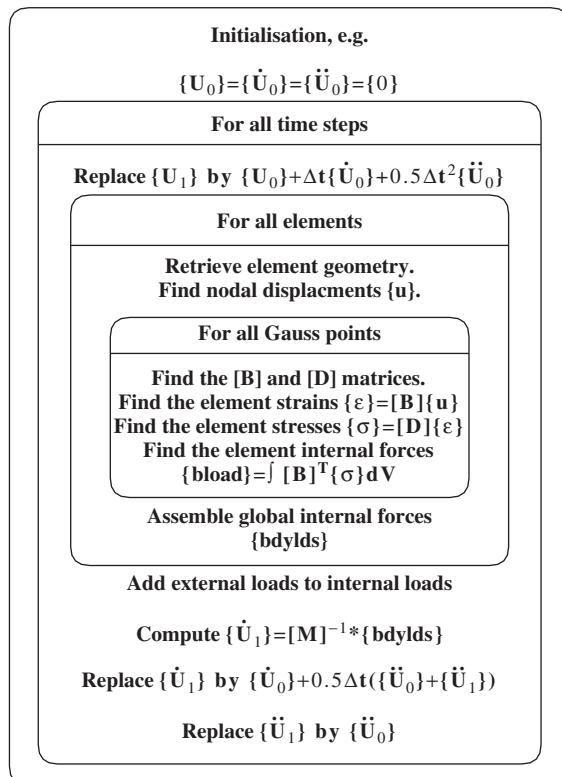
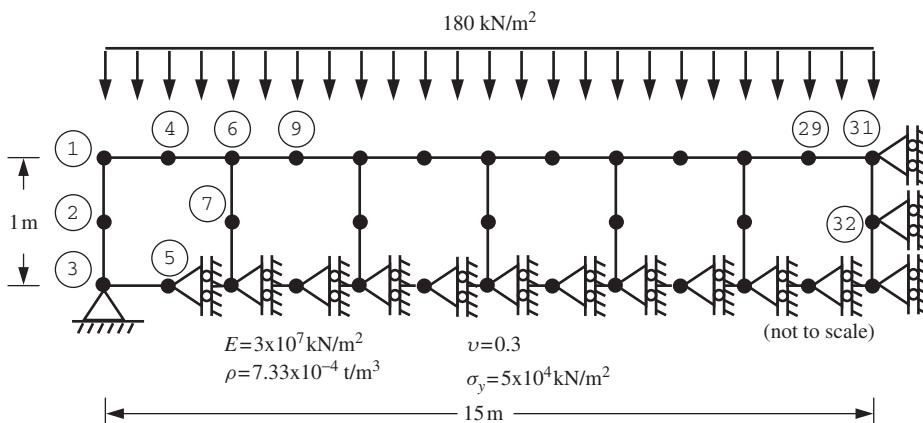


Figure 11.20 Structure chart for Program 11.8

the new solution  $\{U\}_1$  can be computed without solving simultaneous equations at all. Further, the right-hand-side products can again be completed using element-by-element summation and so no global matrices are involved. This procedure is particularly attractive in non-linear problems where the element stiffness  $[k_m]$  is a function of, for example, strain. In the present program, non-linearity is introduced in the form of elastoplasticity, which was described in Chapter 6. The nomenclature used is therefore drawn from the earlier programs in this chapter and from those in Chapter 6, particularly Program 6.1 which dealt with von Mises solids. The structure chart for the program is shown in Figure 11.20 and the problem layout and data in Figure 11.21.



```

nx e   ny e
6      1

np_types
1

prop(e,v,rho,sigy)
3.0e7  0.3  7.33e-4  5.0e4

etype (not needed)

x_coords, y_coords
0.0  2.5  5.0  7.5  10.0  12.5  15.0
0.0 -1.0

dtim    nstep   npri   nres
1.0e-6  10000   50     31

nr,(k,nf(:,k),i=1,nr)
15
3 0 0  5 0 1  8 0 1  10 0 1  13 0 1  15 0 1  18 0 1  20 0 1
23 0 1  25 0 1  28 0 1  30 0 1  31 0 1  32 0 1  33 0 1

loaded_nodes(node(i),val(i,:),i=1,loaded_nodes)
13
1  0.0  0.4167    4  0.0  1.6667    6  0.0  0.8333
9  0.0  1.6667    11 0.0  0.8333   14 0.0  1.6667
16 0.0  0.8333   19 0.0  1.6667   21 0.0  0.8333
24 0.0  1.6667   26 0.0  0.8333   29 0.0  1.6667
31 0.0  0.4167

```

Figure 11.21 Mesh and data for Program 11.8 example

Turning to the program code, after input and initialisation, the von Mises plastic stress-strain matrix  $[\mathbf{D}^p]$  (called  $p1$  in the program) is formed by subroutine `vmdpl1`. The remainder of the program is a large explicit integration time-stepping loop. The displacements (called  $x1$ ) are updated and then, scanning all elements and Gauss points, new strains can be computed. The constitutive relation then determines the appropriate level of stress and hence whether the yield stress has been violated or not. If the yield stress has not been violated, the material remains elastic, otherwise the constitutive matrix is updated by subtracting a proportion of the plastic matrix  $p1$  from the elastic matrix  $\mathbf{d}\mathbf{e}$  (see Figure 6.7). The corrected stresses are then redistributed as ‘body loads’  $b\mathbf{dylds}$ , whence the new accelerations ( $d2x1$ ) can be found and integrated to find the new velocities ( $d1x1$ ). The next cycle of displacements can then be updated. The load weightings are read as data, and the loading function is held in a function subprogram called `load`.

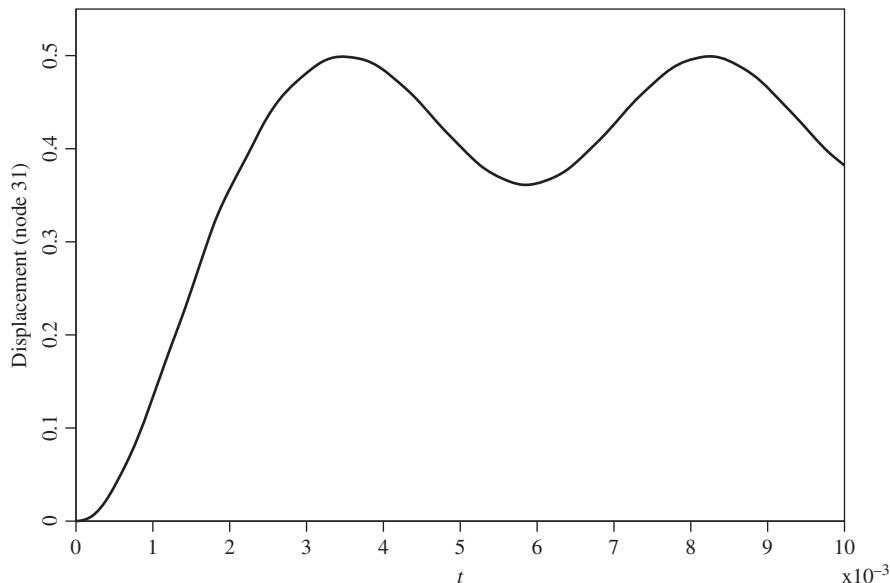
The example problem shown in Figure 11.21 is of a simply supported plane strain slab subjected to a sudden application of a uniformly distributed load of 180 kN/m<sup>2</sup> which remains constant with time. Symmetry has been assumed at the centre of the beam, and along the neutral axis, where only vertical movement is permitted. There are four properties required (`nprops=4`) in this non-linear analysis, namely Young’s modulus  $E$ , Poisson’s ratio  $\nu$ , the mass density  $\rho$  and the (von Mises) yield strength of the material  $\sigma_y$ . The data calls for `nstep=10000` calculation time steps of length `dtim=1.0e-6`. Results are to be printed every `npri=50` steps at node `nres=31`, which lies on the centreline of the beam.

A truncated set of results from the program is printed in Figure 11.22 in the form of elapsed time, load and the  $x$ - and  $y$ -displacements at node 31. Figure 11.23 gives a plot of the centreline displacement of the beam as a function of time computed

There are 50 equations

Result at node 31				
time	load	x-disp	y-disp	
0.0000E+00	-0.1800E+03	0.0000E+00	0.0000E+00	
0.5000E-04	-0.1800E+03	0.0000E+00	-0.2995E-03	
0.1000E-03	-0.1800E+03	0.0000E+00	-0.1214E-02	
0.1500E-03	-0.1800E+03	0.0000E+00	-0.2684E-02	
0.2000E-03	-0.1800E+03	0.0000E+00	-0.4867E-02	
0.2500E-03	-0.1800E+03	0.0000E+00	-0.8084E-02	
0.3000E-03	-0.1800E+03	0.0000E+00	-0.1231E-01	
0.3500E-03	-0.1800E+03	0.0000E+00	-0.1742E-01	
0.4000E-03	-0.1800E+03	0.0000E+00	-0.2331E-01	
0.4500E-03	-0.1800E+03	0.0000E+00	-0.2991E-01	
0.5000E-03	-0.1800E+03	0.0000E+00	-0.3724E-01	
0.5500E-03	-0.1800E+03	0.0000E+00	-0.4494E-01	
0.6000E-03	-0.1800E+03	0.0000E+00	-0.5287E-01	
.				
.				
0.9550E-02	-0.1800E+03	0.0000E+00	-0.4173E+00	
0.9600E-02	-0.1800E+03	0.0000E+00	-0.4127E+00	
0.9650E-02	-0.1800E+03	0.0000E+00	-0.4082E+00	
0.9700E-02	-0.1800E+03	0.0000E+00	-0.4038E+00	
0.9750E-02	-0.1800E+03	0.0000E+00	-0.3997E+00	
0.9800E-02	-0.1800E+03	0.0000E+00	-0.3957E+00	
0.9850E-02	-0.1800E+03	0.0000E+00	-0.3921E+00	
0.9900E-02	-0.1800E+03	0.0000E+00	-0.3887E+00	
0.9950E-02	-0.1800E+03	0.0000E+00	-0.3853E+00	
0.1000E-01	-0.1800E+03	0.0000E+00	-0.3821E+00	

Figure 11.22 Results from Program 11.8 example



**Figure 11.23** Displacement at node 31 vs. time from Program 11.8 example

over the first 10,000 time steps. The development of permanent, plastic deformation is clearly demonstrated.

## 11.2 Glossary of Variable Names

### Scalar integers:

cg_iters	pcg iteration counter
cg_limit	pcg iteration ceiling
i,iel	simple counters
idiag	skyline bandwidth
ifail	warning flag from bisect subroutine
iwp	SELECTED_REAL_KIND(15)
j,jj,k	simple counters
lnode	loaded node number
loaded_nodes	number of loaded nodes
lsense	sense of freedom to be loaded at node lnode
nband	bandwidth of upper triangle
ndof	number of degrees of freedom per element
ndim	number of dimensions
nels	number of elements
neq	number of degrees of freedom in the mesh
nip	number of integrating points
nlen	maximum number of characters in data file basename
nlfp	number of load function points
nln	number of loaded freedoms

nmodes	number of eigenvectors included in superposition
nn	number of nodes
nod	number of nodes per elements
nodof	number of degrees of freedom per node
nof	number of output freedoms
nprops	number of material properties
npri	output printed every npri time steps
np_types	number of different property types
nr	number of restrained nodes
nres	node number at which time history is to be printed
nst	number of stress terms
nstep	number of calculation time steps
nxe, nye	number of columns and rows of elements

**Scalar reals:**

a	real part of complex displacement
aa	working variable
alpha	$\alpha$ from equations (3.22)
area	element area
b	imaginary part of complex displacement
bb	working variable
beta	Newmark time-stepping parameter or $\beta$ from equations (3.22)
cg_tol	pcg convergence tolerance
c1, c2, c3, c4	working constants
det	determinant of Jacobian matrix
dr	damping ratio
dsbar	invariant, $\bar{\sigma}$
dtim	calculation time step
d2, d4, d6	set to 2.0, 4.0 and 6.0
etol	eigenvalue tolerance set to $1 \times 10^{-30}$
f	value of yield function or force vector
fac	measure of yield surface overshoot [ $f$ from (6.35)]
fk	Rayleigh damping parameter on stiffness
fm	Rayleigh damping parameter on mass
fmax	maximum value of yield function $f$ at each time step
fnew	value of yield function after stress increment
f1, f2	working variables
gamma	Newmark time-stepping parameter
k1, k2	working variables
lode_theta	Lode angle, $\theta$
omega	frequency of forcing term
one	set to 1.0
penalty	set to $1 \times 10^{20}$
pt2, pt25, pt5	set to 0.2, 0.25 and 0.5
sigm	mean stress, $\sigma_m$
theta	time-integration weighting parameter
time	holds elapsed time $t$
two	set to 2.0

up                    holds dot product  $\{\mathbf{R}\}_k^T \{\mathbf{R}\}_k$   
zero                set to 0.0

**Scalar character:**

argv                holds data file basename  
element             element type

**Scalar logicals:**

consistent        set to .TRUE. if mass matrix is ‘consistent’  
cg\_converged      set to .TRUE. if pcg process has converged

**Dynamic integer arrays:**

etype	element property type vector
g	element steering vector
g_g	global element steering matrix
g_num	node numbers for all elements
kdiag	diagonal term locations
kdiag_l	diagonal term locations on the left
kdiag_r	diagonal term locations on the right
lf	vector holding loaded freedoms
lp	vector holding output freedoms
nf	nodal freedom matrix
node	output/input nodes vector
num	element node number vector
sense	sense of output nodes

**Dynamic real arrays:**

a	accelerations
acc	accelerations at output freedoms
a1	array holding all loading values at each calculation step
a1	temporary working vector
bbodylds	self-equilibrating global body forces
bee	strain-displacement matrix
bigk	eigenvector matrix
bload	self-equilibrating element body forces
b1	temporary working vector
cm	imaginary part of complex element stiffness matrix
coord	element nodal coordinates
cv	damping matrix
d	displacements or vector used in equation (3.22)
dee	stress-strain matrix
der	shape function derivatives with respect to local coordinates
deriv	shape function derivatives with respect to global coordinates
diag	global lumped mass vector
diag_precon	diagonal preconditioner vector
dis	displacements at output freedoms
d1x0, d1x1	‘old’ and ‘new’ velocities
d2x0, d2x1	‘old’ and ‘new’ accelerations

ecm	used to form element consistent mass matrix
eld	element nodal displacements
ell	element lengths vector
eload	integrating point contribution to bload
eps	strain terms
etensor	holds running total of all integrating point strain terms
fun	shape functions
f1	left-hand-side matrix (stored as a skyline)
g_coord	nodal coordinates for all elements
jac	Jacobian matrix
kd	vector used to set up initial accelerations
kh	global stiffness vector
km	(real part) element stiffness matrix
kp	modified global ‘stiffness’ matrix
ku	global stiffness matrix stored as upper triangle
kv	global stiffness matrix
loads	nodal loads and displacements
mc	global mass matrix used to set up initial accelerations
mm	element mass matrix
mv	global consistent mass matrix
p	‘descent’ vector used in equations (3.22)
pl	plastic $[\mathbf{D}^p]$ matrix (6.29)
points	integrating point local coordinates
prop	element properties matrix
rl	input load function load values
rrmass	vector holding reciprocal of square root of lumped mass
rt	input load function time values
sigma	stress terms
storkm	holds element stiffness matrices
stormm	holds element mass matrices
stress	stress term increments
tensor	holds running total of all integrating point stress terms
u	vector used in equations (3.22)
udiag	transformed and untransformed eigenvectors
v	velocities
val	applied nodal load weightings
vc	vector used to set up initial accelerations
vel	velocities at output freedoms
weights	weighting coefficients
x	‘old’ solution vector
xmod	solutions to modal SDOF equations
xnew	‘new’ solution vector
x0, x1	‘old’ and ‘new’ displacements
x_coords, y_coords	x- and y-coordinates of mesh layout

**Dynamic character array:**

mtype set to ‘l’ for lumped mass and ‘c’ for consistent

**Dynamic complex arrays:**

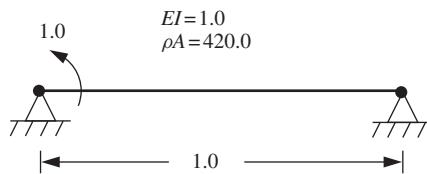
$k_C$	complex element stiffness matrix
loads	complex nodal loads and displacements

**Dynamic character array:**

$mType$	set to 'l' for lumped mass and 'c' for consistent
---------	---

### 11.3 Exercises

1. The undamped beam shown in Figure 11.24 is initially at rest and subjected to a suddenly applied moment of one unit at its left support. Using a single finite element, a time step of 1 s and the constant acceleration method ( $\beta = 1/4$ ,  $\gamma = 1/2$ ), estimate the rotation at both ends of the beam after 2 s.

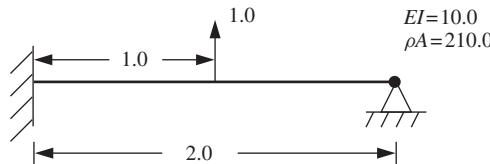
**Figure 11.24**

Answer:  $\theta_1 = 0.284$ ,  $\theta_2 = 0.0036$

2. Repeat the previous question assuming 5% damping. Use Rayleigh damping by assuming the mass matrix damping parameter ( $f_m$ ) equals zero. The fundamental natural frequency of the beam is  $\omega_1 = 0.48$ .

Answer: If  $f_m = 0$ , then  $\zeta_1 = \omega_1 f_k / 2$ , hence with  $\zeta_1 = 0.05$  and  $\omega_1 = 0.48$ ,  $f_k = 0.208$ .  $\theta_1 = 0.254$ ,  $\theta_2 = 0.0015$

3. The undamped propped cantilever shown in Figure 11.25 is initially at rest and subjected to a suddenly applied load at its mid-span. Using two finite elements, a time step of 1 s and the linear acceleration method ( $\beta = 1/6$ ,  $\gamma = 1/2$ ), estimate the deflection under the load after 2 s.

**Figure 11.25**

Answer:  $u = 0.077$

4. The undamped cantilever shown in Figure 11.26 is initially at rest and subjected to a suddenly applied load and moment at its tip. Using one finite element, a time step of 1 s and the constant acceleration method ( $\beta = 1/4$ ,  $\gamma = 1/2$ ), estimate the deflection and rotation at the tip after 2 s.

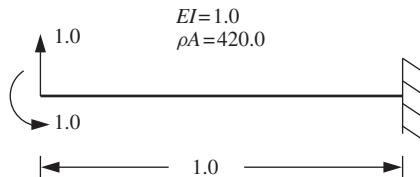


Figure 11.26

Answer:  $u = 0.092$ ,  $\theta = 0.653$

5. A simply supported beam of length  $L = 1$  and properties  $EI = 1.0$ ,  $\rho A = 3.7572$  is subjected to a constant transverse force,  $P = 48$ , which moves across the beam from the left to right support with a constant velocity  $U = 1$ . By discretising the beam into four elements, compute the time-dependent response of the centreline of the beam. Show that the centreline deflection reaches a maximum value that is approximately 1.743 times the deflection that would have been obtained if the load had been placed statically at the centre of the beam. Use equivalent fixed end moments and reactions to model the effect of the moving load at four locations within each element.

Compare your result with the analytical solution at the centreline, given by

$$v = \frac{2PL^3}{\pi^4 EI} \left[ \frac{\sin c\omega_1 t - c \sin \omega_1 t}{1 - c^2} \right]$$

where

$$\omega_1 = \frac{\pi^2}{L^2} \sqrt{\frac{EI}{\rho A}} \quad \text{and} \quad c = \frac{\pi U}{\omega_1 L}$$

## References

- Bathe KJ 1996 *Numerical Methods in Finite Element Analysis*, 3rd edn. Prentice-Hall, Englewood Cliffs, NJ.  
 Key SW 1980 Transient response by time integration: A review of implicit and explicit operators. In *Advances in Structural Dynamics* (ed. Donea J). Applied Science, London, pp. 71–95.  
 Smith IM 1984 Adaptability of truly modular software. *Eng Comput* 1(1), 25–35.  
 Warburton GB 1964 *The Dynamical Behaviour of Structures*. Pergamon Press, Oxford.  
 Wong SW, Smith IM and Gladwell I 1989 PCG methods in transient FE analysis. Part II: Second order problems. *Int J Numer Methods Eng* 28(7), 1567–1576.

# 12

## Parallel Processing of Finite Element Analyses

### 12.1 Introduction

In the previous chapters, serial finite element programs were listed for the solution of a wide variety of problems in engineering and science. As was mentioned in Chapter 1, analyses can be speeded up by vector processing as illustrated in Chapter 5, Program 5.7, but so far vector machines have not been widespread.

The more common approach, used in the majority of supercomputers at the moment, is parallel processing in which many standard (and therefore low cost) multi-core processors are linked together by fast communication networks. About 100,000 cores are typical at the time of writing.

However, supercomputers are still expensive, and an alternative at very low cost is to link together ‘clusters’ of PCs by an ethernet or similar low-cost communications network. Another possibility is to use somebody else’s facility on a ‘pay as you go’ basis, through Cloud Computing.

In this chapter programs are listed which run in parallel on any system capable of supporting MPI (the ‘message passing interface’ standard described in Chapter 1; MPI web reference, 2003). This covers all current supercomputers (vector-parallel, shared memory, distributed memory), PC clusters and some Cloud Computing services. Performance statistics are given for several such systems. OpenMP versions have also been successfully tested but are less portable.

The approach adopted with the MPI programs is to take at least one program from each of the preceding Chapters 5 to 11 and parallelise it. The full range of algorithm types—linear static equilibrium, non-linear static equilibrium, eigenvalue and implicit and explicit transient—are covered, making 10 MPI programs in all.

The methodology assumes the same program running on every ‘MPI process’ of a parallel system (typically one process per core), each process usually operating on different data. From time to time a process needs information that does not reside on the memory associated with that particular process and has to be communicated to it via MPI (Pettipher and Smith, 1997).

**Table 12.1** Effect of mesh subdivision in three dimensions

Mesh subdivision	Number of equations
$10 \times 10 \times 10$	12,580
$20 \times 20 \times 20$	98,360
$40 \times 40 \times 40$	777,520
$50 \times 50 \times 50$	1,514,900
$80 \times 80 \times 80$	6,182,240
$100 \times 100 \times 100$	12,059,800
$400 \times 400 \times 400$	768,959,200
$440 \times 440 \times 440$	1,023,368,720

The benefits sought are both in faster execution times (under perfect conditions  $n$  processes operating in parallel would decrease analysis time by a factor of  $n$ ) and in ability to solve larger problems, because the data can be distributed over the  $n$  processes.

Recent years have seen the increasing adoption of GPUs for accelerating numerical computations and at the end of this chapter, an example program is provided which uses compiler directives to migrate numerically intensive operations from the host CPU to a GPU co-processor.

It is expected that most parallel processing by finite elements will involve problems that are spatially three-dimensional. Data demands for such analyses increase rather dramatically with problem size, as shown in Table 12.1 which refers to an elastic cube meshed by 20-node elements (see Program 12.1). The cube has all four sides on rollers, a fixed base and a free surface.

Thus, although a mesh of  $440 \times 440$  elements in two dimensions might be considered modest,  $440 \times 440 \times 440$  elements in three dimensions would need 8 GB just to store a single vector of equations if the data were not distributed.

Using the program described first in this chapter, a  $440 \times 440 \times 440$  element elastic cube problem was solved in 14 minutes using 4096 MPI processes and in 7 minutes using

**Table 12.2** Serial and parallel program equivalence

Serial version	Parallel version	Remarks
Program 5.6	Program 12.1	20-node brick option, loaded freedoms only
Program 6.13	Program 12.2	3D version, 20-node bricks, Mohr–Coulomb yield criterion
Program 7.5	Program 12.3	3D option
Program 8.5	Program 12.4	3D version
Program 8.6	Program 12.5	3D version
Program 9.2	Program 12.6	3D version
Program 9.5	Program 12.7	3D version
Program 10.3	Program 12.8	3D version
Program 11.7	Program 12.9	3D version
Program 11.8	Program 12.10	3D version
Program 5.7	Program 12.11	3D GPU version

8192 MPI processes. This problem has over 1 billion equations. In the 4096 process run, one or more cores required access to more than 8 GB of memory. This was achieved by using only one core on 4096 16-core processors. Each core on this particular system has shared access to a 16 GB memory bank, giving a typical allocation of 1 GB per core. In this program, memory requirements per MPI process reduce as the number of MPI processes is increased. The 8196 process job used two cores per processor on the same 4096 16-core processor system.

The serial programs closest to their parallel derivatives are given in Table 12.2.

## 12.2 Differences between Parallel and Serial Programs

As far as possible, the parallel programs copy their serial counterparts. For example, comparison of Program 5.6 (serial) and Program 12.1 (parallel) will show that the element integration loop, beginning with label `gauss_pts_1`, is exactly the same in both versions. Such a consistency indicates ‘local’ or ‘embarrassingly parallelisable’ sections of code.

When distributed arrays are involved, for example in the section following ‘pcg equation solution’, the coding is identical with the exception of `_P` in the `DOT_PRODUCT` calls, the `_pp` appendage to array names (`r` becomes `r_pp` and so on) and the distributed convergence check `checon_par`.

In what follows, the differences between parallel and serial programs are described. These differences are common to all the parallelised programs that use MPI.

### 12.2.1 Parallel Libraries

Serial libraries `new_library` and `geometry` perform the same tasks as `main` and `geom` in the earlier chapters but are augmented by 11 others, as shown in Table 12.3.

**Table 12.3** Parallel libraries

Library name	Usage
<code>precision</code>	Sets precision for <code>REAL</code> variables
<code>mp_interface</code>	Initialises and finalises MPI
<code>timing</code>	Routines to assist with performance evaluation
<code>global_variables</code>	Designation of some widely used variables as ‘global’, not declared elsewhere
<code>gather_scatter</code>	MPI routines for collecting data from, or distributing it to, parallel processors
<code>input</code>	MPI routines for reading input data
<code>output</code>	MPI routines for writing results
<code>maths</code>	MPI routines for distributed mathematical operations
<code>loading</code>	MPI routines for distributing loads
<code>eigen</code>	MPI routines for eigenvalue problems
<code>mpi_wrapper</code>	Switch compilation between serial and parallel mode (optional)
<code>mpi_stubs</code>	Dummy MPI library for serial mode (optional)

An optional library `mpi_wrapper` builds the parallel libraries and programs in ‘serial mode’ using `mpi_stubs`, a dummy MPI library supplied with the book.

### 12.2.2 Global Variables

In the serial programs, all variables were declared in all programs. In the parallel versions, some widely used variables are declared as ‘global’. These are listed with their meanings in Table 12.4.

These variables must not be additionally declared.

### 12.2.3 MPI Library Routines

The same MPI routines are used in all programs. There are only a dozen or so that are listed below with their purposes (see Appendix F for further details of subroutines used in this chapter and their arguments):

---

<code>SHUTDOWN</code>	Finalise MPI: must appear
<code>DOT_PRODUCT_P</code>	Distributed version of dot product
<code>SUM_P</code>	Distributed version of array SUM
	<i>Note:</i> We take the liberty of using capitals as if these were part of FORTRAN
<code>max_p</code>	Finds maximum of a distributed integer variable
<code>find_pe_procs</code>	Finds how many processes are being used
<code>calc_nels_pp</code>	Finds number of elements per process (variable)
<code>calc_neq_pp</code>	Finds number of equations per process (variable)
<code>make_ggl</code>	Builds distributed $g$ vectors (see Section 3.7.10 for description of $g$ )
<code>gather</code>	See Section 12.2.8
<code>scatter</code>	See Section 12.2.8
<code>checon_par</code>	Convergence check for distributed vectors
<code>reindex</code>	See Section 12.2.9
<code>read_pxxx</code>	See Section 12.2.6

---

**Table 12.4** Global variables

---

<code>nels_pp</code>	Number of elements per process (variable)
<code>neq</code>	Number of equations
<code>neq_pp</code>	Number of equations per process (variable)
<code>ntot</code>	Total number of degrees of freedom per element
<code>ielp_e</code>	Global counter for element
<code>iel_start</code>	Starting element number on the local process
<code>ieq_start</code>	Starting equation number on the local process
<code>numpe</code>	ID number (rank) of the local process
<code>npes</code>	Total number of processes used
<code>ier</code>	MPI error flag

---

### 12.2.4 The \_pp Appendage

Distributed arrays and their upper bounds carry the appendage `_pp`. A difference from the serial programs is that it is more convenient to begin array addresses at 1 rather than 0. So, the serial `p(0:neq)` becomes `p_pp(neq_pp)` in parallel.

### 12.2.5 Simple Test Problems

Most of the programs in this chapter allow users to import their own externally generated models. However, in order to assess the benefits of parallelism it is necessary to be able to refine meshes readily for the same basic analysis. Therefore, a simple mesh generating program `p12meshgen` is provided which can generate input decks for simple cuboidal geometries of varying sizes. Note that `p12meshgen` is a serial program.

For example, to generate an input deck for Program 12.1, the mesh generator requires file `p121.mg` which is similar to the data files used in earlier chapters. A `<base_name>.mg` file is provided for most of the programs in this chapter (for an example, see Figure 12.8). The first variable `program_name` identifies the parallel program. The second variable `iotype` allows the user to select between two data formats:

```
"parafem"           ! format for the parallel programs
"paraview"          ! format for visualisation in ParaView
```

The program is executed using the command:

```
p12meshgen <base_name>
p12meshgen p121      ! if the input file is called p121.mg
```

The programs in this chapter read the ParaFEM format and `iotype` is set to `parafem` by default. Note that displacements are output in the Ensight Gold format. If ParaView is to be used for visualisation, it is necessary to rerun `p12meshgen` with `iotype` set to `paraview` in order to generate an additional set of geometry files in the ParaView supported format. The files output by `p12meshgen` are listed with descriptions in Tables 12.5 and 12.6.

The mesh-generating program `p12meshgen` uses problem-specific routines to generate boundary conditions (see Appendix F).

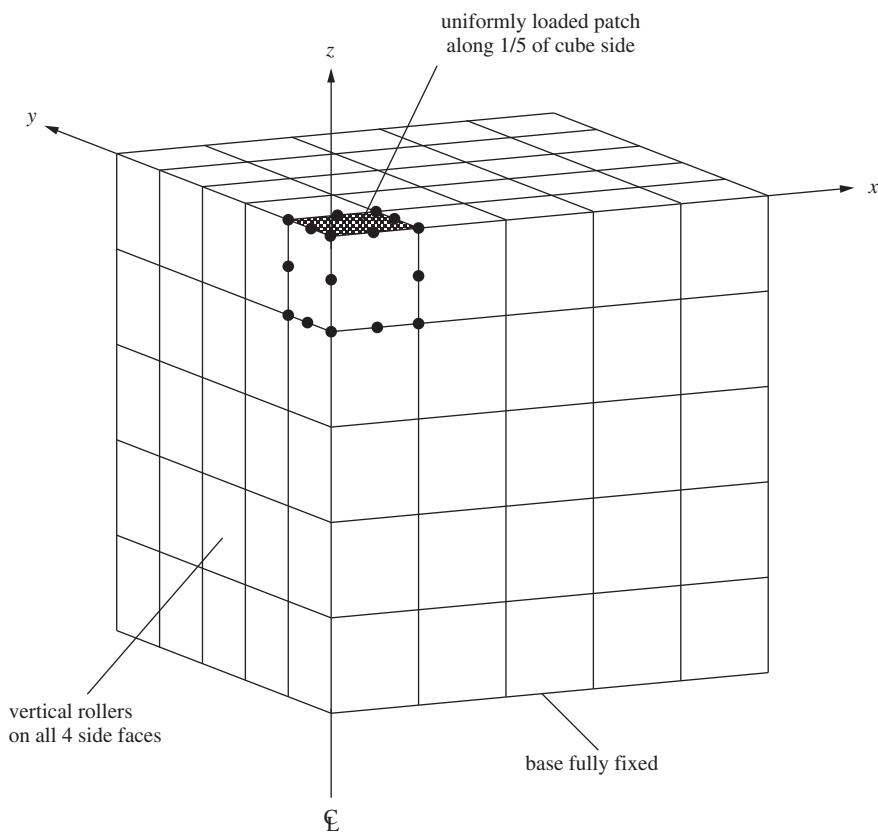
For example, Programs 12.1 and 12.2, respectively analyse a cuboid of elastic or elasto-plastic material made up from 20-node bricks as shown in Figure 12.1.

**Table 12.5** List of ParaFEM files

<code>&lt;base_name&gt;.dat</code>	Basic program control data required by each program
<code>&lt;base_name&gt;.d</code>	Nodal coordinates and element steering array
<code>&lt;base_name&gt;.bnd</code>	Boundary conditions (restraint array rest)
<code>&lt;base_name&gt;.lds</code>	Loaded nodes (optional)
<code>&lt;base_name&gt;.fix</code>	Fixed freedoms (optional)
<code>&lt;base_name&gt;.lid</code>	Nodes with fixed velocities (Program 12.6 only)
<code>&lt;base_name&gt;.res</code>	<i>Output summary results</i>

**Table 12.6** List of ParaView Files

<code>&lt;base_name&gt;.ensi.case</code>	Header file listing associated files
<code>&lt;base_name&gt;.ensi.geo</code>	Nodal coordinates and element steering array
<code>&lt;base_name&gt;.ensi.ndbnd</code>	Boundary conditions (restraint array rest)
<code>&lt;base_name&gt;.ensi.ndlds</code>	Loaded nodes (optional)
<code>&lt;base_name&gt;.ensi.ndfix</code>	Fixed nodes (optional)
<code>&lt;base_name&gt;.ensi.displ-&lt;step&gt;</code>	<i>Output displacements</i>
<code>&lt;base_name&gt;.ensi.vel-&lt;step&gt;</code>	<i>Output velocities</i> (Program 12.6)
<code>&lt;base_name&gt;.ensi.ndttr-&lt;step&gt;</code>	<i>Output temperatures</i> (Program 12.4)
<code>&lt;base_name&gt;.ensi.ndptl-&lt;step&gt;</code>	<i>Output potentials</i> (Programs 12.3 and 12.5)
<code>&lt;base_name&gt;.ensi.eigv-&lt;step&gt;</code>	<i>Output eigen vectors</i> (Program 12.8)

**Figure 12.1** Mesh for Programs 12.1 and 12.2

Note that the  $z$ -axis is the vertical rather than the  $y$ -axis as in Program 5.6. All four vertical faces are on rollers; the front and left are planes of symmetry and the back and right are external boundaries. The base is completely fixed and the top completely free (except at the roller edges).

To simplify the loading, it is assumed to occupy a square patch extending to one-fifth of the cube surface in the  $x$ - and  $y$ -directions. It is therefore assumed that numbers

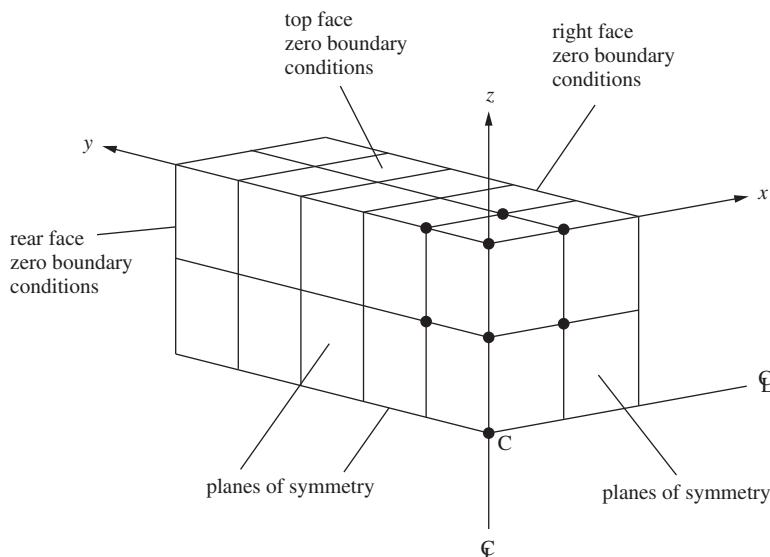
of elements in the  $x$ - and  $y$ -directions are multiples of 5. In `p12meshgen`, subroutine `cube_bc20` applies the appropriate boundary conditions and subroutine `loading` the appropriate loading.

Programs 12.3, 12.4 and 12.5 analyse a cuboidal box of 8-node elements as shown in Figure 12.2.

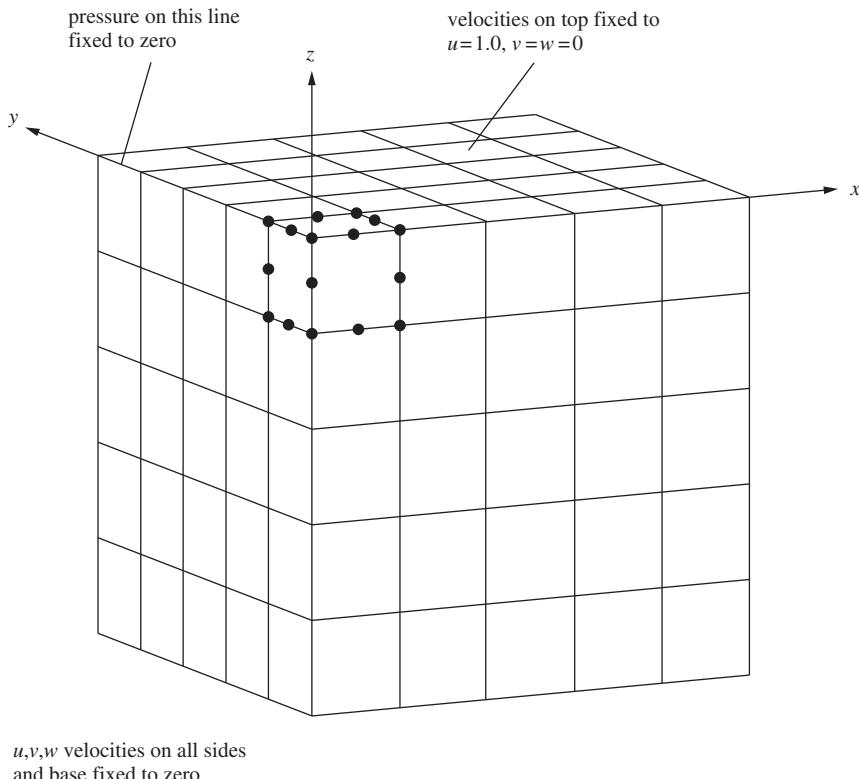
The boundary conditions assumed are that the top, back and right-hand faces of the box have zero potential (or temperature) while the left, front and base are planes of symmetry. In Program 12.3 temperature or heat flux (see Program 7.5) can be specified at C, the centre of the box, and the results are printed for the centre and a few nodes to the right. In Programs 12.4 and 12.5, which involve transient analyses, the variation of temperature or pressure with time is printed at C only, for a given (uniform) initial distribution. Subroutine `box_bc8` applies the appropriate boundary conditions.

Program 12.6 conducts a Navier–Stokes analysis for the classic lid-driven cavity which is assumed to be cuboidal (no symmetry in this case). Thus, the velocities on all faces are fixed to zero in  $x$ ,  $y$  and  $z$ , except for the top face that is driven with constant velocity in the  $x$ -direction but otherwise fixed. To avoid a singularity in the pressure field, pressure is assumed to be zero along the left-hand edge of the top of the box (analogous to Programs 9.1 and 9.2, where the top left-hand corner had zero pressure). Subroutine `ns_cube_bc20` applies the appropriate boundary conditions. A typical mesh is shown in Figure 12.3.

Program 12.7 analyses coupled consolidation of a cuboid of porous elastic material. The boundary conditions on the solid part are the same as those in Figure 12.1 while the fluid pressure is zero on the top surface only. Subroutine `biot_loading` again assumes a square patch extending to one-fifth of the surface edge length. Subroutine `biot_cube_bc20` supplies the appropriate boundary conditions. This program is the only one that does not read in externally generated geometries. Instead, it shows how to



**Figure 12.2** Mesh for Programs 12.3, 12.4, 12.5



**Figure 12.3** Mesh for Program 12.6

generate meshes for simple geometries (in parallel) within the program itself. A typical mesh is shown in Figure 12.4.

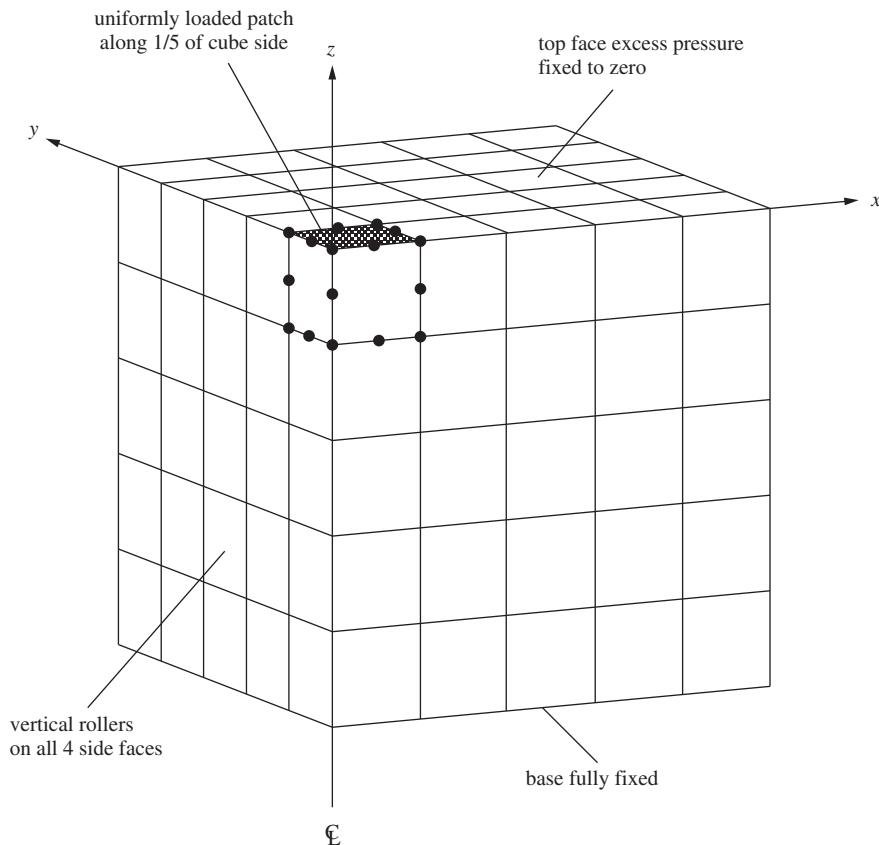
The remaining programs in the chapter, Programs 12.8, 12.9 and 12.10, all analyse cuboidal cantilevers of elastic or elastoplastic material as shown in Figures 12.5 and 12.6.

The elements can be 8-node (Program 12.8) or 20-node (Programs 12.9, 12.10) bricks and the front  $x-z$  face is completely fixed in  $x$ ,  $y$  and  $z$ . In Program 12.9, a distributed load is applied along the edge indicated by the arrow in Figure 12.6. For Program 12.10, an even number of elements in the  $x$ -direction is assumed, so that the load can be applied at the mid-point of the top of the free end face as shown in Figure 12.6.

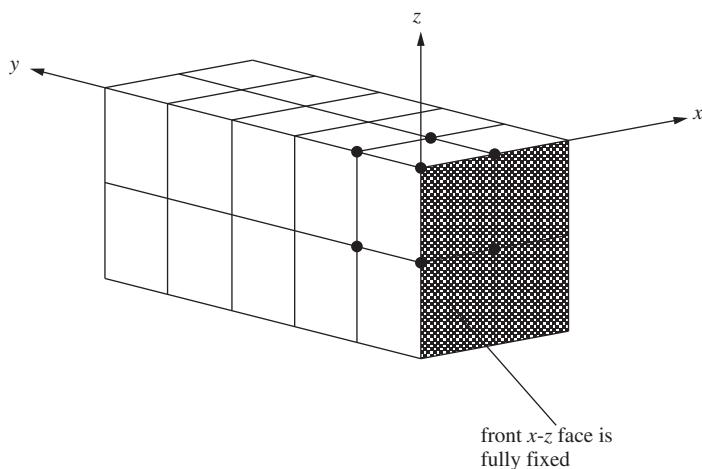
### 12.2.6 Reading and Writing

The simple approach adopted here is that data are read, and results written, by a single (not necessarily the same) process. Data, having been read by one process, are then broadcast to all other processes by MPI routines such as `read_p121`. These are unique to each parallel program as the `_p121` appendage implies.

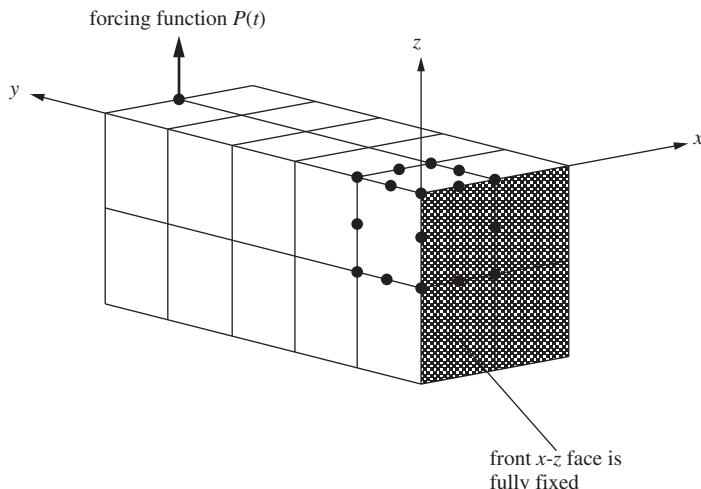
For summary results to be written easily, it is necessary to find which process contains the desired quantities. In the programs in this chapter a single output process is identified and used.



**Figure 12.4** Mesh for Program 12.7



**Figure 12.5** Mesh for Program 12.8



**Figure 12.6** Mesh for Programs 12.9 and 12.10

For visualisation purposes, results for quantities at all the nodes are written using the MPI routine `dismsh_ensi_p`. Again, writing is done by one process, with remote data being collected from other processes with the help of MPI routines `calc_nodes_pp` and `scatter_nodes`. Separate files are created for each step where output is required over multiple load or time steps.

### 12.2.7 *rest Instead of nf*

In the serial programs a ‘node freedom array’, `nf` (see Section 3.7.10) was employed. This array contained information about every node in the mesh, whether restrained or not. In very large problems this is wasteful because the restrained (usually boundary) nodes become a smaller and smaller proportion of the total. For this reason the parallel programs use an array `rest` instead of `nf`. Exactly the same restraint information is created as would be the case for `nf`.

Program `p12meshgen` generates `rest` using the boundary condition routines described in the previous section and then writes it to the file `<base_name>.bnd`. Thus routines like `cube_bc20` return `rest` from a knowledge of the number of elements in each direction and the problem-specific boundary conditions. The MPI programs in this chapter read in `rest` using `read_rest`, then rearrange it using subroutines `rearrange` or `rearrange_2` before it is used in routines to calculate the steering vector `g` from the restraint data.

Because there is now no `nf`, replacements for subroutine `num_to_g` which was used in all serial programs are necessary. The replacement routine is `find_g3` in Programs 12.1, 12.2, 12.6, 12.7, `find_g4` in Programs 12.3, 12.4, 12.5 and `find_g` in Programs 12.8, 12.9, 12.10. The different versions are necessary because rather large volumes of data are being searched.

### 12.2.8 Gathering and Scattering

This is done very neatly in the serial programs using the power of modern FORTRAN as described in Section 1.9.4. Thus a typical gather-matrix multiply-scatter loop around the elements, the core of EBE iteration methods, reads

```

elements_2: DO iel=1,nels
    g=g_g(:,iel);  pmul=p(g)
    utemp=MATMUL(km,pmul);  u(g)=u(g)+utemp
END DO elements_2

```

In parallel,  $u$  and  $p$  are distributed as  $u_{\text{pp}}$  and  $p_{\text{pp}}$  while  $utemp$  and  $pmul$  are distributed as two-dimensional arrays  $utemp_{\text{pp}}$  and  $pmul_{\text{pp}}$  that hold all the components of  $utemp$  and  $pmul$  for that processor.

Thus the parallel loop becomes

```

CALL gather(p_pp,pmul_pp)
elements_2: DO iel=1,nels_pp
    utemp_pp(:,iel)=MATMUL(km,pmul_pp(:,iel))
END DO elements_2
CALL scatter(u_pp,utemp_pp)

```

### 12.2.9 Reindexing

When loads are read in or displacements fixed, their global node or freedom numbers are specified. In parallel, the appropriate equations are distributed across the processes in some way and so the appropriate indexing must be found. The MPI routine `load` does this behind the scenes and populates a distributed loads vector. For fixed quantities where the node, sense and value data are provided, first the corresponding global equation number is identified by `find_no` and then its local position is determined using `reindex`.

### 12.2.10 Domain Composition

For parallel FE processing, pieces of a large mesh have to be allocated to the processes. These pieces are traditionally called ‘subdomains’. It is also traditional to speak of the whole mesh or ‘domain’ being ‘decomposed’ into its constituent subdomains. This use of words implies that a large mesh (domain) is assembled, in principle by the global assembly techniques described in previous chapters, and that the resulting global equations are ‘decomposed’ or torn apart into smaller pieces. Indeed, some early implementations of this process were called ‘diakoptics’, implying a cutting up procedure. ‘Substructuring’, and ‘block’ and ‘frontal’ methods are similar variants, the main application being the solution, by elimination techniques, of the very large sets of linear algebraic equations which govern the static equilibrium of linear and non-linear FE systems.

All of this suggests the parallelisation of the ‘global’ strategy used in earlier chapters of this book. When Gaussian elimination methods are used, elimination can proceed independently on equations relating to a particular subdomain, stored on a particular process, until the boundaries of the subdomain are reached. Then communication is necessary

between processes. This can be a rather complicated procedure, and a large literature has developed around the theme of optimising subdomain distributions. Often equations are solved directly within subdomains but iteratively at the boundaries.

In contrast, in this book a simple approach is used, based on the element-by-element methods used in previous chapters. In these, no global equation system matrices are ever constructed and therefore it is more meaningful to speak of ‘domain composition’ rather than ‘decomposition’. But of course subdomains have to be identified.

In the element-by-element technique, see Section 3.5, the essential operations that involve inter-processor communication are the ‘gather, matrix multiply, scatter’ procedure typified by equations (3.23) and the dot products typified by equations (3.22), (3.28) and (3.29). Clearly, different subdomain distributions will affect the amount of communication involved. Margetts (2002) (see also Smith and Margetts, 2003) has shown that there is certainly no simple solution to this problem for complicated domains. Some compositions lead to fewer large messages being exchanged between processes and others to more short messages.

### 12.2.11 Third-party Mesh-partitioning Tools

For the cuboidal meshes used in this chapter, the programs use, by default, a ‘naive’ composition by slices on  $x-z$  planes. However, the programs can easily be adapted to use compositions created by third-party ‘mesh-partitioning’ tools. This usually requires manipulation of data formats and here, scripts are provided that enable the mesh partitioner METIS (Karypis and Kumar, 1999) to be used with the programs in this chapter. The procedure to follow is outlined below. Note that the tools only support 4-node tetrahedra and 8-node hexahedra.

#### (i) Download and install METIS

METIS can be downloaded from <http://glaros.dtc.umn.edu/gkhome/>. Build instructions are given on the website for LINUX/UNIX-based systems only.

#### (ii) Convert geometry to METIS format

A simple script pf2metin converts the ParaFEM geometry data to the METIS format:

```
pf2metin <geometry_parafem>.d <geometry_metis>.d
```

For example, the METIS file p121\_metis.d is created from the ParaFEM file p121.d by

```
pf2metin p121.d p121_metis.d
```

#### (iii) Run METIS

Working in the same directory as the model files, the METIS program partnmesh partitions geometry\_metis.d into nparts, the number of partitions (or processes) required:

```
partnmesh <geometry_metis>.d <nparts>
```

Here, p121\_metis is partitioned for execution on eight processes:

```
partnmesh p121_metis.d 8
```

Note that METIS outputs two partition files, one with the suffix `eprt#` and the other `nprt#` (where # is the number of partitions). Only the `eprt#` file is used next.

#### (iv) Convert partitioned geometry to ParaFEM format

Finally, the partitioned geometry is converted back into ParaFEM format using `metout2pf`:

```
metout2pf <partitionfile> <basenamein> <basenameout>
```

where `basenamein` is the name of the original data set (without the file suffix) and `basenameout` is the name used for the partitioned data set. Thus the command

```
metout2pf p121_metis.d.eprt.8 p121 p121_8
```

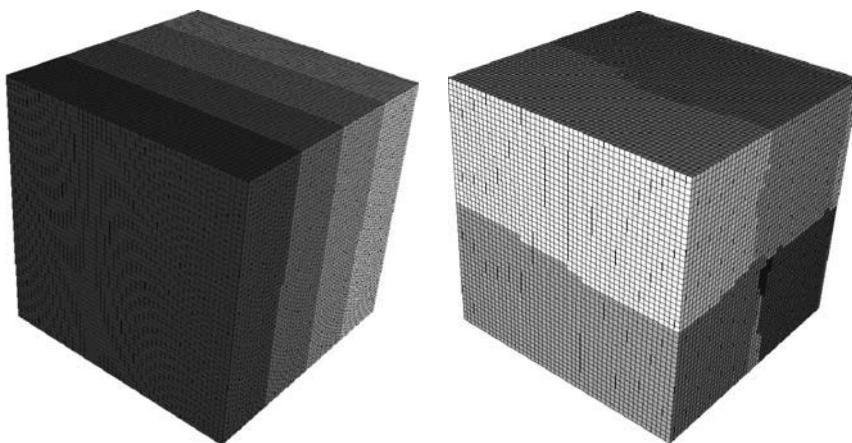
creates the new data set `p121_8.dat`, `p121_8.d`, `p121_8.bnd`, `p121_8.lds` and `p121_8.psize`.

The `psize` file holds the value of `nels_pp` for each MPI process and is read into the programs in this chapter by routine `calc_nels_pp` if `partitioner==2`. If `partitioner==1`, the program uses the internal naive partitioning strategy and the `psize` file is not required.

The ‘naive’ and METIS-generated domain compositions, shown in Figure 12.7 for the case of a simple cubic domain, led to broadly similar analysis times when executed using the same number of processes. The ‘naive’ domains are on the left of the figure and the METIS domains are on the right.

### 12.2.12 Load Balancing

In our ‘naive’ domain compositions the ‘load’ on processes (the amount of computation they do) is almost perfectly distributed or ‘balanced’ by assigning almost equal numbers of elements, `nels_pp` and equations, `neq_pp` to each process. In contrast, the number



**Figure 12.7** Mesh compositions for four processes

of elements in the METIS partitions is more variable. A stronger weighting is given to minimising the boundaries between subdomains than to load balancing. Note that if the mesh is too small for the number of processes requested, the analysis will not continue.

We are now in a position to describe the parallel programs in detail.

## Program 12.1 Three-dimensional analysis of an elastic solid. Compare Program 5.6

```

PROGRAM p121
!-----
!      Program 12.1 three dimensional analysis of an elastic solid
!      using 20-node brick elements, preconditioned conjugate gradient
!      solver; diagonal preconditioner diag_precon; parallel version
!      loaded_nodes only
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE steering; USE new_library; IMPLICIT NONE
! neq,ntot are now global variables - must not be declared
INTEGER,PARAMETER::nodof=3,ndim=3,nst=6
INTEGER::loaded_nodes,iel,i,j,k,iter,limit,nn,lr,nip,nod,nel,ndof,      &
          npes_pp,node_end,node_start,nodes_pp,meshgen,partitioner,nlen
REAL(iwp),PARAMETER::zero=0.0_iwp
REAL(iwp)::e,v,det,tol,up,alpha,beta,q; LOGICAL::converged=.false.
CHARACTER(LEN=50)::argv; CHARACTER(LEN=15)::element; CHARACTER(LEN=6)::ch
!----- dynamic arrays -----
REAL(iwp),ALLOCATABLE::points(:,:,:),dee(:,:,:),weights(:),val(:,:),
  disp_pp(:),g_coord_pp(:,:,:,:),jac(:,:,:),der(:,:,:),deriv(:,:),
  bee(:,:,:),storkm_pp(:,:,:,:),eps(:),sigma(:),diag_precon_pp(:),
  p_pp(:),r_pp(:),&
  x_pp(:),xnew_pp(:),u_pp(:),pmul_pp(:,:,:),utemp_pp(:,:),
  d_pp(:),&
  timest(:),diag_precon_tmp(:,:,:),eld_pp(:,:,:),temp(:))
INTEGER,ALLOCATABLE::rest(:,:,:),g_num_pp(:,:,:),g_g_pp(:,:),
  node(:)
!----- input and initialisation -----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p121(argv,numpe,e,element,limit,loaded_nodes,meshgen,nel,
  & nodof,nip,nn,nod,lr,partitioner,tol,v)
CALL calc_nels_pp(argv,nel,npes,numpe,partitioner,nel,
  & ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod, nel),g_coord_pp(nod,ndim,nel),
  & rest(nr,nodof+1)); g_num_pp=0; g_coord_pp=zero; rest=0
CALL read_g_num_pp(argv,iel_start,nn,npes,numpe,g_num_pp)
IF(meshgen == 2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
CALL read_rest(argv,numpe,rest); timest(2)=elap_time()
ALLOCATE(points(nip,ndim),dee(nst,nst),jac(ndim,ndim),der(ndim,nod),
  & deriv(ndim,nod),bee(nst,ntot),weights(nip),eps(nst),sigma(nst),
  & storkm_pp(ntot,ntot,nel),pmul_pp(ntot,nel),
  & utemp_pp(ntot,nel),g_g_pp(ntot,nel))
!----- find the steering array and equations per process -----
CALL rearrange(rest); g_g_pp=0; neq=0
elements_0: DO iel=1,nel
  CALL find_g3(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_0
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp

```

```

CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
ALLOCATE(p_pp(neq_pp),r_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp),          &
         u_pp(neq_pp),d_pp(neq_pp),diag_precon_pp(neq_pp)); diag_precon_pp=zero
p_pp=zero; r_pp=zero; x_pp=zero; xnew_pp=zero; u_pp=zero; d_pp=zero
!----- element stiffness integration and build the preconditioner -----
dee=zero; CALL deemat(dee,e,v); CALL sample(element,points,weights)
storkm_pp=zero
elements_1: DO iel=1,nels_pp
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); jac=MATMUL(der,g_coord_pp(:,:,iel))
        det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
        CALL beemat(bee,deriv)
        storkm_pp(:,:,:,iel)=storkm_pp(:,:,:,iel) +
            MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i) &
        END DO gauss_pts_1
    END DO elements_1
ALLOCATE(diag_precon_tmp(ntot,nels_pp)); diag_precon_tmp=zero
elements_2: DO iel=1,nels_pp ; DO i=1,ndof
    diag_precon_tmp(i,iel) = diag_precon_tmp(i,iel)+storkm_pp(i,i,iel)
END DO; END DO elements_2
CALL scatter(diag_precon_pp,diag_precon_tmp); DEALLOCATE(diag_precon_tmp)
IF(numpe==1) THEN
    OPEN(11,FILE=argv(1:nlen)//".res",STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I7,A)') "This job ran on ",npes," processes"
    WRITE(11,'(A,3(I12,A))') "There are ",nn," nodes", nr, &
        " restrained and ",neq," equations"
    WRITE(11,'(A,F10.4)') "Time to read input is:",timest(2)-timest(1)
    WRITE(11,'(A,F10.4)') "Time after setup is:",elap_time()-timest(1)
END IF
!----- get starting r -----
IF(loaded_nodes>0) THEN
    ALLOCATE(node(loaded_nodes),val(ndim,loaded_nodes)); node=0; val=zero
    CALL read_loads(argv,numpe,node,val)
    CALL load(g_g_pp,g_num_pp,node,val,r_pp(1:)); q=SUM_P(r_pp(1:))
    IF(numpe==1) WRITE(11,'(A,E12.4)') "The total load is:",q
    DEALLOCATE(node,val)
END IF
DEALLOCATE(g_g_pp); diag_precon_pp=1._iwp/diag_precon_pp
d_pp=diag_precon_pp*r_pp; p_pp=d_pp; x_pp=zero
!----- preconditioned cg iterations -----
iters=0; timest(3)=elap_time()
iterations: DO
    iters=iters+1; u_pp=zero; pmul_pp=zero; utemp_pp=zero
    CALL gather(p_pp,pmul_pp)
    elements_3: DO iel=1,nels_pp
        utemp_pp(:,:,:,iel) = MATMUL(storkm_pp(:,:,:,iel),pmul_pp(:,:,:,iel))
        CALL dgemv('n',ntot,ntot,1.0_iwp,storkm_pp(:,:,:,iel),ntot,
                  pmul_pp(:,:,:,iel),1,0.0_iwp,utemp_pp(:,:,:,iel),1) &
    END DO elements_3 ;CALL scatter(u_pp,utemp_pp)
!----- pcg equation solution -----
    up=DOT_PRODUCT_P(r_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
    xnew_pp=x_pp+p_pp*alpha; r_pp=r_pp-u_pp*alpha
    d_pp=diag_precon_pp*r_pp; beta=DOT_PRODUCT_P(r_pp,d_pp)/up
    p_pp=d_pp+p_pp*beta; CALL checon_par(xnew_pp,tol,converged,x_pp)
    IF(converged.OR.iters==limit)EXIT
END DO iterations
IF(numpe==1) THEN
    WRITE(11,'(A,I6)') "The number of iterations to convergence was ",iters

```

```

      WRITE(11,'(A,F10.4)')"Time to solve equations was :",
     & elap_time()-timest(3)
      WRITE(11,'(A,E12.4)')"The central nodal displacement is :",xnew_pp(1)
END IF
DEALLOCATE(p_pp,r_pp,x_pp,u_pp,d_pp,diag_precon_pp,storkm_pp,pmul_pp)
!----- recover stresses at centroidal gauss point -----
ALLOCATE(eld_pp(ntot,nels_pp)); eld_pp=zero; points=zeros; nip=1; iel=1
CALL gather(xnew_pp(1:),eld_pp); DEALLOCATE(xnew_pp)
IF(numpe==1)WRITE(11,'(A)')"The Centroid point stresses for element 1 are"
gauss_pts_2: DO i=1,nip
   CALL shape_der(der,points,i); jac=MATMUL(der,g_coord_pp(:,:,iel))
   CALL invert(jac); deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
   eps=MATMUL(bee,eld_pp(:,iel)); sigma=MATMUL(dee,eps)
   IF(numpe==1.AND.i==1) THEN
      WRITE(11,'(A,I5)')"Point ",i ; WRITE(11,'(6E12.4)') sigma
   END IF
END DO gauss_pts_2; DEALLOCATE(g_coord_pp)
!----- write out displacements -----
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
IF(numpe==1) THEN; WRITE(ch,'(I6.6)') numpe
   OPEN(12,file=argv(1:nlen)//".ensi.DISPL"//ch,status='replace',
        action='write')
   WRITE(12,'(A)') "Alya Ensight Gold --- Vector per-node variable file"
   WRITE(12,'(A/A/A)') "part", " 1","coordinates"
END IF
ALLOCATE(disp_pp(nodes_pp*ndim),temp(nodes_pp)); disp_pp=zero; temp=zero
CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,ndim,nodes_pp,
                   node_start,node_end,eld_pp,disp_pp,1)
DO i=1,ndim ; temp=zero
   DO j=1,nodes_pp; k=i+(ndim*(j-1)); temp(j)=disp_pp(k); END DO
   CALL dismsh_ensi_p(12,1,nodes_pp,npes,numpe,1,temp)
END DO ; IF(numpe==1) CLOSE(12)
IF(numpe==1) WRITE(11,'(A,F10.4)')"This analysis took :",
     & elap_time()-timest(1)
CALL SHUTDOWN()
END PROGRAM p121

```

In contrast to programs in earlier chapters, the first step in running the analysis is to generate the full input deck using the simple mesh-generating program p12meshgen (Section 12.2.5). This reads the data provided in file p121.mg (listed in Figure 12.8) and outputs p121.dat, p121.d, p121.bnd and p121.lds. If the mesh-partitioning tool METIS is used to compose the domains (Section 12.2.11), the input deck will have the additional file p121.psize.

After declarations, timing is started. Next, the number identifying each MPI process (its ‘rank’), numpe and the total number of processes used for the analysis, npes is found using find\_pe\_procs. Basic program data are read by the first process from file p121.dat and then broadcast to all other processes using read\_p121.

The number of elements per process may be calculated internally by calc\_nels\_pp. However, if partitioner==2, an external mesh-partitioning program has been used and calc\_nels\_pp reads nels\_pp for each process from p121.psize. After nels\_pp has been determined, arrays g\_num\_pp, g\_coord\_pp and rest are allocated.

```

program
'p121'

iotype    nels   nxe   nze   nod   nip
'parafem' 64000  40    40    20    8

aa      bb      cc      e      v
0.25  0.25  0.25 100.0  0.3

tol     limit
1.0E-5 200

```

**Figure 12.8** Data for Program 12.1 example

Externally generated global node numbers are read from file p121.d using `read_g_num_pp`, which distributes the required data to each process. Note that compared with the serial version, `g_num` contains element data for all elements `nels`. Here `g_num_pp` only stores data for its own elements `nels_pp`. If `meshgen==2`, `abaqus2sg` rearranges the element node numbering to follow the numbering convention used in the book. The nodal coordinates are read from p121.d using `read_g_coord_pp` and the restraint array `rest` is read from file p121.bnd using `read_rest`. After reading geometry data, the main arrays are allocated and `rest` is rearranged.

Loop `elements_0` replaces the serial loop `elements_1`, and finds the steering array `g_g_pp`. The largest equation on each process is found using the FORTRAN array intrinsic `MAXVAL`. The parallel routine `max_p` determines the total number of equations `neq` in the problem. The numbers of equations to be distributed to each process can then be calculated using `calc_neq_pp`. The inter-process communications tables, which are required by `gather` and `scatter`, can then be built by `make_ggl`. The distributed equation arrays can also be allocated.

The section commented ‘element stiffness integration etc’ down to END DO `gauss_pts_1` is identical in parallel and serial versions, but building the diagonal preconditioner in parallel involves a scatter operation and is moved outside the loop.

Information about the analysis is written by the first process. The next section of coding relocates the global loading `val` entries (read from file p121.lds by `read_loads`) to the appropriate processors using `load`. Next, the program prints out the total load `q` and inverts the preconditioner `diag_precon_pp`.

The section commented ‘preconditioned cg iterations’ is the parallel equivalent of the similarly annotated section in Program 5.6, involving gather and scatter as described in Section 12.2.8. In a similar way the section commented ‘pcg equation solution’ mirrors the serial version in an obvious way. Only the centreline vertical displacement of the elastic cuboid is printed. This part of the program uses external subroutine `dgemv` (see Appendix G).

The stress recovery section, involving the loop labelled `gauss_pts_2`, uses exactly the same coding as the serial version but the stresses are only printed for the central surface element at the first Gauss point.

Finally, displacements are written to a file in the Ensight Gold format by `dismsh_ensi_p` for use in ParaView as described in Chapter 1. Only the first process writes the results and routines `calc_nodes_pp` and `scatter_nodes` are used to collect nodal values from remote processes.

The example analysed is an elastic cube with a uniform pressure of 25 units on a square patch at the centre of the cube. Data are listed as Figure 12.8 and results as Figure 12.9. The vertical deflection is seen to be  $-0.8571$  units and the vertical stress under the load  $-24.98$  (compared to  $-25.00$  applied).

In all, the parallel program is about 50% longer than its serial counterpart. Two salient aspects of performance are shown in Figures 12.10 and 12.12. The success of iterative methods clearly depends on the number of iterations for convergence as a proportion of problem size (Smith and Wang, 1998). Figure 12.10 shows that the iteration count tends to decrease sharply with problem size, and this is illustrated graphically in Figure 12.11.

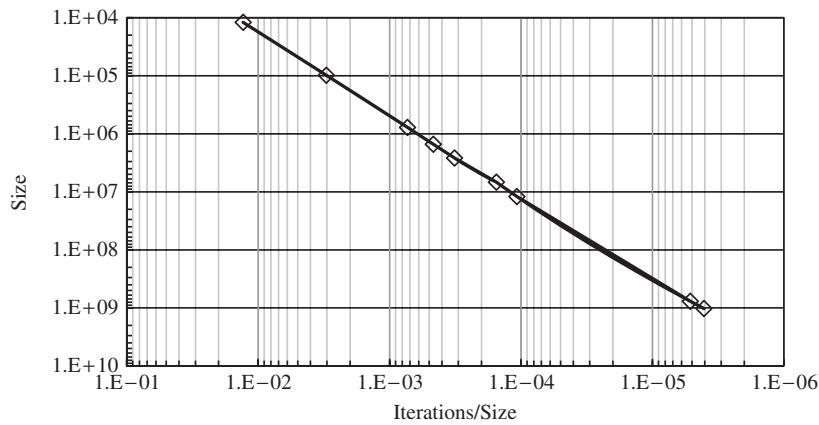
Figure 12.12 shows, for a given problem size, the analysis time on a ‘supercomputer’ decreasing but levelling off when a sufficient number of processes has been reached. Thus for any given problem size there comes a point where adding extra processes brings no benefit. In some cases, adding more processes makes the job run more slowly again. This limiting number of processes increases with problem size. Conversely, a problem can be so small that parallelisation does not bring any benefit at all in terms of analysis time. However, in some cases, distribution of data may allow parallel processing of a job that could not be run at all serially due to memory limitations. Figure 12.13 shows a typical plot generated using ParaView. Contours are of magnitude of displacement.

```
This job ran no      32 processes
There are 270641 nodes 24161 restrained and 777520 equations
Time to read input is:    1.1561
Time after of setup is:   1.7041
The number of iterations to convergence was     569
Time to solve equations was : 22.3174
The central nodal displacement is : -0.8571E+00
The Centroid point stresses for element 1 are
Point      1
-0.1657E+02 -0.1657E+02 -0.2498E+02  0.1636E-02  0.6623E-02  0.6623E-02
This analysis took   : 24.8256
```

**Figure 12.9** Results from Program 12.1 example

Problem Size	Iterations to convergence	Iters/Size
12,000	156	1.30E-2
98,000	297	3.03E-3
777,000	568	7.31E-4
1,514,000	704	4.65E-4
2,613,000	838	3.21E-4
6,812,000	1049	1.54E-4
12,059,000	1297	1.07E-4
768,959,200	3963	5.15E-6
1,023,368,720	4152	4.06E-6

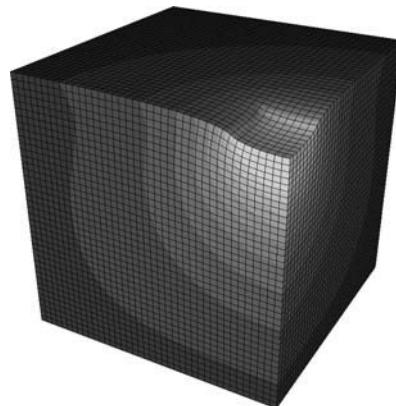
**Figure 12.10** Iteration to convergence against problem size: Program 12.1



**Figure 12.11** Iterations/size vs. problem size: Program 12.1 (Cray XE6)

Mesh	No of Processes	Analysis Time (secs)
40x40x40	8	96
	16	48
	32	25
	64	14
	128	8.0
	256	5.5
100x100x100	16	486
	32	256
	64	140
	128	83
	256	106
400x400x400	1024	2721
	2048	1213
	4096	662

**Figure 12.12** Performance statistics: Program 12.1 (Cray XE6)



**Figure 12.13** Deformed mesh for 64,000-element problem: Program 12.1

## Program 12.2 Three-dimensional analysis of an elastoplastic (Mohr-Coulomb) solid. Compare Program 6.13

```

PROGRAM p122
!-----
!     Program 12.2 3-d strain of an elastic-plastic (Mohr-Coulomb) solid
!     viscoplastic strain method; pcg parallel
!     pick up current x not x = .0; load or displacement control
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE new_library; IMPLICIT NONE
!neq,ntot are now global variables - must not be declared
INTEGER,PARAMETER::nodof=3,nst=6,ndim=3
INTEGER::nn,nod,nr,nip,i,j,k,iel,plasiters,nels,ndof,node_end,nodes_pp, &
plasits,cjitters,cjits,cjtots,incs,iy,loaded_nodes,node_start,nlen, &
partitioner,meshgen,npes_pp,fixed_freedoms,fixed_freedoms_pp, &
fixed_freedoms_start
LOGICAL::plastic_converged,cj_converged; CHARACTER(LEN=50)::argv
CHARACTER(LEN=15)::element='hexahedron'; CHARACTER(LEN=6)::ch
REAL(iwp)::e,v,det,phi,c,psi,dt,f,dsbar,dq1,dq2,dq3,lode_theta,presc, &
sigm,pi,snph,plastol,cjtol,up,alpha,beta,big
REAL(iwp),PARAMETER::zero=0.0_iwp,penalty=1.e20_iwp
!----- dynamic arrays -----
REAL(iwp),ALLOCATABLE::loads_pp(:,points(:,:)),bdyllds_pp(:,valf(:)), &
evpt_pp(:,:,pmul_pp(:,:,dee(:,:,jac(:,:,weights(:),store_pp(:), &
oldis_pp(:),der(:,:,deriv(:,:,bee(:,:,eld(:),eps(:),ld0_pp(:), &
sigma(:),bload(:),eload(:),erate(:),evp(:),devp(:),
m1(:,:,m2(:,:,m3(:,:,flow(:,:,storkm_pp(:,:,:),r_pp(:),temp(:), &
tensor_pp(:,:,:),stress(:),totd_pp(:),qinc(:),p_pp(:),x_pp(:), &
xnew_pp(:),u_pp(:),utemp_pp(:,:)),diag_precon_pp(:),d_pp(:),disp_pp(:), &
diag_precon_tmp(:,:),timest(:),g_coord_pp(:,:,:),val(:,:)
INTEGER,ALLOCATABLE::rest(:,:,no_f(:),g_num_pp(:,:,nodef(:), &
g_g_pp(:,:,node(:),sense(:),no_pp_temp(:),no_f_pp(:)
!----- input and initialisation -----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p122(argv,numpe,c,cjits,cjtol,e,element,fixed_freedoms, &
loaded_nodes,incs,meshgen,nels,nip,nn,nod,lr,phi,partitioner,plasits, &
plastol,psi,v); IF(fixed_freedoms==0) fixed_freedoms_pp=0
CALL calc_nels_pp(argv,nels,npes,numpe,partitioner,nels_pp)
ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod,nels_pp),g_coord_pp(nod,ndim,nels_pp), &
rest(nr,no_f+1)); g_num_pp=0; g_coord_pp=zero; rest=0
CALL read_g_num_pp(argv,iel_start,nn,npes,numpe,g_num_pp)
IF(meshgen == 2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
CALL read_rest(argv,numpe,rest); timest(2)=elap_time()
ALLOCATE(points(nip,ndim),weights(nip),m1(nst,nst),qinc(incs),eps(nst), &
dee(nst,nst),evpt_pp(nst,nip,nels_pp),m2(nst,nst),m3(nst,nst),evp(nst), &
tensor_pp(nst,nip,nels_pp),g_g_pp(ntot,nels_pp),utemp_pp(ntot,nels_pp), &
stress(nst),storkm_pp(ntot,ntot,nels_pp),jac(ndim,ndim),der(ndim,nod), &
deriv(ndim,nod),bee(nst,ntot),eld(ntot),pmul_pp(ntot,nels_pp), &
sigma(nst),bload(ntot),eload(ntot),erate(nst),devp(nst),flow(nst,nst))
!----- find the steering array and equations per process -----

```

```

CALL read_qinc(argv,numpe,qinc); CALL rearrange(rest); g_g_pp=0; neq=0
elements_1: DO iel=1,nels_pp
    CALL find_g(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_1; neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
IF (numpe==1) THEN
    OPEN(11,FILE=argv(1:nlen)//'.res',STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I5,A)')"This job ran on ", npes, " processes"
    WRITE(11,'(A,3(I7,A))')"There are ",nn," nodes",nr,
        &
        " restrained and ",neq," equations"
    WRITE(11,'(A,F10.4)')"Time after setup is:",elap_time()-timest(1)
END IF
ALLOCATE(loads_pp(neq_pp),bdylds_pp(neq_pp),oldis_pp(neq_pp),
        &
        r_pp(neq_pp),totd_pp(neq_pp),p_pp(neq_pp),x_pp(neq_pp),ld0_pp(neq_pp),&
        u_pp(neq_pp),diag_precon_pp(neq_pp),d_pp(neq_pp),xnew_pp(neq_pp))
totd_pp=zero;tensor_pp=zero; p_pp=zero; xnew_pp=zero; diag_precon_pp=zero
oldis_pp=zero;ld0_pp=zero; pi=ACOS(-1._iwp); snph=SIN(phi*pi/180._iwp)
dt=4._iwp*(1._iwp+v)*(1._iwp-2._iwp*v)/(e*(1._iwp-2._iwp*snph*snph))
IF (numpe==1) WRITE(11,'(A,E12.4)')"The critical timestep is ",dt
!---- element stiffness integration, preconditioner & set initial stress--
CALL deemat(dee,e,v); CALL sample(element,points,weights); storkm_pp=zero
tensor_pp=zero
elements_2: DO iel=1,nels_pp
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i)
        jac=MATMUL(der,g_coord_pp(:,:,iel)); det=determinant(jac)
        CALL invert(jac); deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        storkm_pp(:,:,:,iel)=storkm_pp(:,:,:,iel)+&
            MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1
END DO elements_2
ALLOCATE(diag_precon_tmp(ntot,nels_pp)); diag_precon_tmp=zero
elements_2a: DO iel=1,nels_pp ; DO i=1,ndof
    diag_precon_tmp(i,iel) = diag_precon_tmp(i,iel)+storkm_pp(i,i,iel)
END DO; END DO elements_2a
CALL scatter(diag_precon_pp,diag_precon_tmp); DEALLOCATE(diag_precon_tmp)
!----- invert preconditioner -----
IF (loaded_nodes>0) THEN
    ALLOCATE(node.loaded_nodes),val(ndim,loaded_nodes)); val=zero; node=0
    CALL read_loads(argv,numpe,node,val)
    CALL load(g_g_pp,g_num_pp,node,val,ld0_pp(1:))
END IF
IF (fixed Freedoms>0) THEN
    ALLOCATE(nodef(fixed Freedoms),no_pp_temp(fixed Freedoms),
        &
        no_f(fixed Freedoms),sense(fixed Freedoms),valf(fixed Freedoms))
    nodef=0; no_f=0; no_pp_temp=0; sense=0; valf=zero
    CALL read_fixed(argv,numpe,nodef,sense,valf)
    CALL find_no(nodef,rest,sense,no_f)
    CALL reindex(ieq_start,no_f,no_pp_temp,fixed Freedoms_pp,&
        fixed Freedoms_start,neq_pp)
    ALLOCATE(no_f_pp(fixed Freedoms_pp),store_pp(fixed Freedoms_pp))
    no_f_pp=0; store_pp=0; no_f_pp=no_pp_temp(1:fixed Freedoms_pp)
    DEALLOCATE(nodef,no_f,sense,no_pp_temp)
END IF
IF (fixed Freedoms_pp>0) THEN
    DO i=1,fixed Freedoms_pp; j=no_f_pp(i)-ieq_start+1
        diag_precon_pp(j)=diag_precon_pp(j)+penalty
        store_pp(i)=diag_precon_pp(j)

```

```

    END DO
END IF; diag_precon_pp=1._iwp/diag_precon_pp
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
ALLOCATE(disp_pp(nodes_pp*ndim),temp(nodes_pp)); disp_pp=zero; temp=zero
!----- load increment loop -----
load_increments: do iy=1,incs
  plasiters=0; bdylds_pp=zero; evpt_pp=zero; cjtots=0
  IF(numpe==1) WRITE(11,'(/,A,I5)') "Load Increment ",iy
!----- plastic iteration loop -----
plastic_iterations: DO ; plasiters=plasiters+1; loads_pp=zero
  IF(plasiters==1) THEN
    IF(fixed_freedoms_pp>0) THEN
      DO i=1,fixed_freedoms_pp; j=no_f_pp(i)-ieq_start+1
        k=fixed_freedoms_start+i-1
        loads_pp(j)=store_pp(i)*valf(k)*qinc(iy)
      END DO
    END IF
    IF(loader_nodes>0) THEN
      loads_pp=ld0_pp*qinc(iy); loads_pp=loads_pp+bdylds_pp
    END IF
  ELSE
    IF(loader_nodes>0) loads_pp=ld0_pp*qinc(iy)
    loads_pp=loads_pp+bdylds_pp
    IF(fixed_freedoms_pp>0)THEN
      DO i=1,fixed_freedoms_pp
        j=no_f_pp(i)-ieq_start+1; loads_pp(j)=zero
      END DO
    END IF
  END IF
!---- if x=0 p and r are just loads but in general p=r=loads-A*x -----
r_pp=zero; CALL gather(x_pp,pmul_pp)
elements_2b: DO iel=1,nels_pp
  utemp_pp(:,iel)=MATMUL(storkm_pp(:,:,iel),pmul_pp(:,iel))
END DO elements_2b; CALL scatter(r_pp,utemp_pp)
r_pp=loads_pp-r_pp; d_pp=diag_precon_pp*r_pp; p_pp=d_pp; cjters=0
!----- solve the simultaneous equations by pcg -----
conjugate_gradients: DO ; cjters=cjters+1; u_pp=zero; pmul_pp=zero
  CALL gather(p_pp,pmul_pp)
  elements_3: DO iel=1,nels_pp
    utemp_pp(:,iel)=MATMUL(storkm_pp(:,:,iel),pmul_pp(:,iel))
  END DO elements_3; CALL scatter(u_pp,utemp_pp)
  IF(fixed_freedoms_pp>0) THEN
    DO i=1,fixed_freedoms_pp; j=no_f_pp(i)-ieq_start+1
      IF(plasiters==1) THEN; u_pp(j)=p_pp(j)*store_pp(i)
      ELSE; u_pp(j)=zero; END IF
    END DO
  END IF
!----- pcg process -----
  up=DOT_PRODUCT_P(r_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
  xnew_pp=x_pp+p_pp*alpha; r_pp=r_pp-u_pp*alpha
  d_pp=diag_precon_pp*r_pp; beta=DOT_PRODUCT_P(r_pp,d_pp)/up
  p_pp=d_pp+p_pp*beta; cj_converged=.TRUE.
  CALL checon_par(xnew_pp,cjtol,cj_converged,x_pp)
  IF(cj_converged.or.cjters==cjits) EXIT
END DO conjugate_gradients
cjtot=cjtot+cjters; loads_pp=xnew_pp; pmul_pp=zero
!----- check plastic convergence -----
CALL checon_par(loads_pp,plastol,plastic_converged,oldis_pp)

```

```

IF(plasiters==1)plastic_converged=.FALSE.
IF(plastic_converged.OR.plasiters==plasits)bbodylds_pp=zero
CALL gather(loads_pp,pmul_pp); utemp_pp=zero
elements_4: DO iel=1,nels_pp
    bload=zero; eld=pmul_pp(:,iel)
    gauss_points_2: DO i=1,npip
        CALL shape_der(der,points,i); jac=MATMUL(der,g_coord_pp(:,:,iel))
        det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
        CALL beemat(bee,deriv); eps=MATMUL(bee,eld)
        eps=eps-evpt_pp(:,i,iel); sigma=MATMUL(dee,eps)
        stress=sigma+tensor_pp(:,i,iel)
        CALL invar(stress,sigm,dsbar,lode_theta)
!----- check whether yield is violated -----
        CALL mocouf(phi,c,sigm,dsbar,lode_theta,f)
        IF(plastic_converged.OR.plasiters==plasits)THEN
            devp=stress
        ELSE
            IF(f>=zero)THEN
                CALL mocouq(psi,dsbar,lode_theta,dq1,dq2,dq3)
                CALL formm(stress,m1,m2,m3); flow=f*(m1*dq1+m2*dq2+m3*dq3)
                erate=MATMUL(flow,stress); evp=erate*dt
                evpt_pp(:,i,iel)=evpt_pp(:,i,iel)+evp; devp=MATMUL(dee,evp)
            END IF
        END IF
        IF(f>=zero)THEN
            eload=MATMUL(TRANSPOSE(bee),devp)
            bload=bload+eload*det*weights(i)
        END IF
        IF(plastic_converged.OR.plasiters==plasits)      &
            tensor_pp(:,i,iel)=stress
    END DO gauss_points_2
!----- compute the total bodyloads vector -----
    utemp_pp(:,iel)=utemp_pp(:,iel)+bload
END DO elements_4; CALL scatter(bbodylds_pp,utemp_pp)
IF(plastic_converged.OR.plasiters==plasits)EXIT
END DO plastic_iterations; totd_pp=totd_pp+loads_pp
IF(numpe==1)THEN
    WRITE(11,'(A,E12.4)')"The displacement is ",totd_pp(1)
    WRITE(11,'(A)')" sigma z     sigma x     sigma y"
    WRITE(11,'(3E12.4)')tensor_pp(3,1,1),tensor_pp(1,1,1),tensor_pp(2,1,1)
    WRITE(11,'(A,I12)')"The total number of cj iterations was ",cjtot
    WRITE(11,'(A,I12)')"The number of plastic iterations was ",plasiters
    WRITE(11,'(A,F11.2)')"cj iterations per plastic iteration were ",  &
        REAL(cjtot)/REAL(plasiters)
END IF
!----- write out the displacements for ParaView -----
IF(numpe==1)THEN; WRITE(ch,'(I6.6)') iy
    OPEN(12,file=argv(1:nlen)//".ensi.DISPL-"/ch,status='replace',      &
        action='write')
    WRITE(12,'(A)') "Alya Ensight Gold --- Vector per-node variable file"
    WRITE(12,'(A/A/A)') "part", " 1","coordinates"
END IF
disp_pp=zero; utemp_pp=zero; CALL gather(totd_pp(1:),utemp_pp)
CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,ndim,nodes_pp,      &
    node_start,node_end,utemp_pp,disp_pp,1)
DO i=1,ndim; temp=zero
    DO j=1,nodes_pp; k=i+(ndim*(j-1)); temp(j)=disp_pp(k); END DO
    CALL dismsh_ensi_p(12,1,nodes_pp,npes,numpe,1,temp)

```

```

END DO; IF(numpe==1) CLOSE(12)
IF(plasiters==plasits) EXIT
END DO load_increments
IF(numpe==1)WRITE(11,'(A,F10.4)')"This analysis took: ", &
  elap_time()-timest(1); CALL SHUTDOWN()
END PROGRAM p122

```

This program follows naturally from the previous one by extending the material behaviour beyond the elastic range. The post-elastic region is perfectly plastic and governed by the Mohr–Coulomb failure criterion that was first introduced in Program 6.3 and applied in three-dimensional analyses in Program 6.13. The example problem, created by p12meshgen (Section 12.2.5), is a cuboidal mesh with the central loaded patch a vertical uniformly distributed load extending to one-fifth of the surface in the  $x$ - and  $y$ -directions ( $z$  is vertical). Thus only multiples of 5 should be used for  $nxe$  in this simple case.

After INTEGER and REAL data have been read and broadcast by `read_p122`, load increment array `qinc` (see Programs 6.1 and 6.2) is read and broadcast by `read_qinc`. The loop labelled `elements_1` can be compared with the same loop in Program 6.13. In parallel, the loop is used to find  $g$  for each element only as the geometry is now read in from external files.

The loop labelled `elements_2`, with its embedded integration loop `gauss_pts_1`, carries over essentially unaltered from serial to parallel versions.

This program supports displacement control in addition to load control. Data for fixed displacements, `nodef`, sense and `valf`, are read by the first process and broadcast to the other processes using `read_fixed`. The corresponding equation numbers `no` are found on each process using `find_no` and distributed to `no_pp` using `reindex`.

In Program 6.13 each set of conjugate gradient iterations begins with a starting  $x$  value of zero. In large non-linear problems it is beneficial to use the value of  $x$  computed in the previous step as the starting  $x$  for a new step. Typically this halves the number of pcg iterations in subsequent steps, and this procedure is used in this program (see loop `elements_2b`).

Loops labelled `conjugate_gradients`, `elements_3` and `elements_4` with its embedded `gauss_points_2` are equivalent in serial and parallelised versions.

Each load increment `iy` outputs a file containing all nodal displacements in the Ensight Gold format using `dismesh_ensi_p`. Results for the full loading sequence can be animated in ParaView.

The problem analysed as shown in Figure 12.1 with data in Figure 12.14, deals with the bearing capacity of a square uniformly loaded punch at the surface of an elastoplastic block. Since  $\phi$  (phi) is set to zero, the Mohr–Coulomb criterion reduces to the Tresca. There is no analytical solution to this problem, but approximate solutions indicate a value in 3D a little higher than the  $(2 + \pi)c_u$  recorded in plane strain.

In our case the 3D ultimate load is computed to be about  $5.20c_u$ , as shown in the results listed in Figure 12.15. Performance data are shown in Figure 12.16.

In contrast to the programs of earlier chapters, most programs in this chapter can read in externally generated models with arbitrary geometries. Figure 12.17 shows a typical plot for a dinosaur trackway simulation using Program 12.2. The displacement control capability is used to indent a ‘rigid’ three-toed raptor foot into a ‘sandy soil’. In the figure, contours map the magnitude of vertical displacement and arrows indicate direction of soil movement.

```

program
'p122'

iotype    nels   nxe   nze   nod   nip
'parafem' 8000   20    20    20    8

aa   bb   cc   incs
0.5  0.5  0.5  12

phi   c     psi   e       v
0.0   100.0  0.0   10000.0  0.3

plasits  cjits  plastol  cjtol
1000      1000    1.0E-4   1.0E-5

incs(qinc(i),i=1,incs
200.0 100.0 50.0 50.0 50.0 30.0
20.0 10.0 10.0 5.0 5.0 5.0

```

**Figure 12.14** Data for Program 12.2 example

This job ran on 16 processes  
 There are 35721 nodes 6081 restrained and 98360 equations  
 Time after setup is: 1.3561  
 The critical timestep is 0.5200E-01

```

Load Increment 1
The displacement is -0.6856E+01
  sigma z   sigma x   sigma y
  -0.2003E+0. -0.1423E+03 -0.1423E+03
The total number of cj iterations was      298
The number of plastic iterations was      2
cj iterations per plastic iteration were 149.00

Load Increment 2
The displacement is -0.1031E+02
  sigma z   sigma x   sigma y
  -0.2993E+03 -0.2124E+03 -0.2124E+03
The total number of cj iterations was      620
The number of plastic iterations was      9
cj iterations per plastic iteration were 68.89

.
.

Load Increment 11
The displacement is -0.3935E+02
  sigma z   sigma x   sigma y
  -0.5065E+03 -0.3395E+03 -0.3395E+03
The total number of cj iterations was      13893
The number of plastic iterations was      243
cj iterations per plastic iteration were 57.17

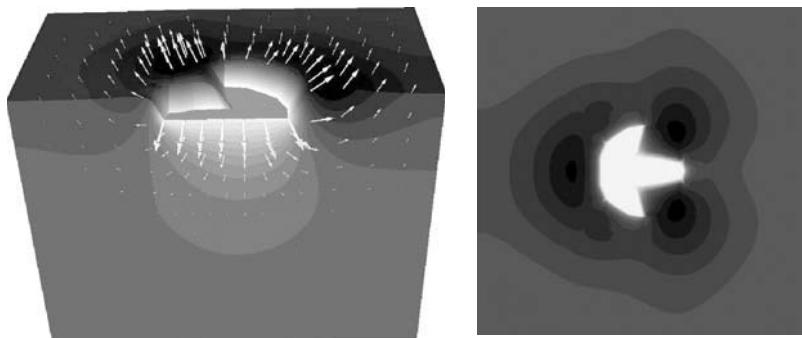
Load Increment 12
The displacement is -0.4106E+02
  sigma z   sigma x   sigma y
  -0.5208E+03 -0.3367E+03 -0.3367E+03
The total number of cj iterations was      30568
The number of plastic iterations was      1000
cj iterations per plastic iteration were 30.57
This analysis took: 589.9529

```

**Figure 12.15** Results from Program 12.2 example

Mesh	No of Processes	Analysis Time (secs)
20x20x20	16	590
	32	321
	64	180
	128	113

**Figure 12.16** Performance statistics: Program 12.2 (Cray XE6)



**Figure 12.17** Typical plot for dinosaur track simulation: Program 12.2

Parallelised versions of programs in Chapter 6 have been used to study various aspects of dinosaur trackways. For example, Falkingham *et al.* (2009) demonstrate that a particular fossil track, believed to have been made by a dinosaur with webbed feet, could have been made by a dinosaur without webbing. Zones of plastic failure between the toes can appear to be imprints of webbed feet. In another paper, the programs are used to demonstrate that ‘apparent’ poor biodiversity in fossil track sites could be in part due to soil mechanics. Simply put, lightweight animals may not leave tracks where heavier ones do (Falkingham *et al.*, 2011).

These studies required meshing arbitrary geometries in 3D. With large element counts and non-linear material behaviour, this work has benefited from parallel processing.

### Program 12.3 Three-dimensional Laplacian flow. Compare Program 7.5

```
PROGRAM p123
!-----
! Program p12.3 3D analysis of Laplace's equation using 8-node bricks,
!               preconditioned conjugate gradient solver diagonal
!               preconditioner; parallel; externally generated model
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE new_library; IMPLICIT NONE
!neq,ntot are now global variables - not declared
INTEGER, PARAMETER::ndim=3,nodof=1
INTEGER::nod,nn,nr,nip,i,j,k,iters,limit,iel,partitioner,meshgen, &
```

```

node_end,node_start,nodes_pp,loaded_freedoms,fixed_freedoms,
fixed_freedoms_pp,fixed_freedoms_start,nlen,nres,is,it,
loaded_freedoms_pp,loaded_freedoms_start,nels,ndof,npes_pp
REAL(iwp),PARAMETER::zero=0.0_iwp,penalty=1.e20_iwp
REAL(iwp)::kx,ky,kz,det,tol,up,alpha,beta,q; CHARACTER(LEN=6)::ch
CHARACTER(LEN=15)::element; CHARACTER(LEN=50)::argv
LOGICAL::converged=.false.
REAL(iwp),ALLOCATABLE::points(:,:),weights(:,elid_pp(:,:)),kay(:,:),
fun(:,jac(:,der(:,deriv(:,col(:,row(:,kcx(:,kcy(:,kcz(:,diag_precon_pp(:,p_pp(:,r_pp(:,x_pp(:,xnew_pp(:,u_pp(:,pmul_pp(:,utemp_pp(:,d_pp(:,val(:,diag_precon_tmp(:,store_pp(:,storkc_pp(:,elid(:,timest(:,val_f(:,g_coord_pp(:,ptl_pp(:,
INTEGER,ALLOCATABLE::rest(:,g_num_pp(:,g_g_pp(:,no(:,no_pp(:,no_f_pp(:,no_pp_temp(:,sense(:,node(:),
!-----input and initialisation-----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p123(argv,numpe,element,fixed_freedoms,kx,ky,kz,limit,
loaded_freedoms,meshgen,nels,nip,nn,nod,nr,nres,partitioner,tol)
CALL calc_nels_pp(argv,nels,npes,numpe,partitioner,nels_pp)
ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod,nels_pp),g_coord_pp(nod,ndim,nels_pp))
g_num_pp=0; g_coord_pp=zero
CALL read_g_num_pp(argv,iel_start,nn,npes,numpe,g_num_pp)
IF(meshgen == 2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
IF (nr>0) THEN; ALLOCATE(rest(nr,nodof+1)); rest=0
CALL read_rest(argv,numpe,rest); END IF
ALLOCATE (points(nip,ndim),jac(ndim,ndim),storkc_pp(ntot,ntot,nels_pp), &
deriv(ndim,nod),kcx(ntot,ntot),weights(nip),der(ndim,nod), &
pmul_pp(ntot,nels_pp),utemp_pp(ntot,nels_pp),col(ntot,1),eld(ntot), &
g_g_pp(ntot,nels_pp),kcy(ntot,ntot),row(1,ntot),kczi(ntot,ntot))
!----- find the steering array and equations per process -----
timest(2)=elap_time(); g_g_pp=0; neq=0
IF(nr>0) THEN; CALL rearrange_2(rest)
elements_0: DO iel=1, nels_pp
    CALL find_g4(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_0
ELSE
    g_g_pp=g_num_pp !When nr = 0, g_num_pp and g_g_pp are identical
END IF
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN; it=numpe; is=i; END IF; END DO
IF(numpe==it)THEN
    OPEN(11,FILE=argv(1:nlen)//'.res',STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I5,A)')"This job ran on ", npes, " processes"
    WRITE(11,'(A,3(I7,A))')"There are ",nn," nodes",nr,
    " restrained and ",neq," equations"
    WRITE(11,'(A,F10.4)')"Time after setup is ",elap_time()-timest(1)
END IF
ALLOCATE(p_pp(neq_pp),r_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp), &
u_pp(neq_pp),diag_precon_pp(neq_pp),d_pp(neq_pp))
r_pp=zero; p_pp=zero; x_pp=zero; xnew_pp=zero; diag_precon_pp=zero
!----- element stiffness integration and storage -----
CALL sample(element,points,weights); storkc_pp=zero
elements_1: DO iel=1,nels_pp

```

```

kcx=zero; kcy=zero; kcz=zero
gauss_pts_1: DO i=1,nip
    CALL shape_der (der,points,i); jac=MATMUL(der,g_coord_pp(:,:,iel))
    det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
    row(1,:)=deriv(1,:); eld=deriv(1,:); col(:,1)=eld
    kcx=kcx+MATMUL(col,row)*det*weights(i); row(1,:)=deriv(2,:)
    eld=deriv(2,:); col(:,1)=eld
    kcy=kcy+MATMUL(col,row)*det*weights(i); row(1,:)=deriv(3,:)
    eld=deriv(3,:); col(:,1)=eld
    kcz=kcz+MATMUL(col,row)*det*weights(i)
END DO gauss_pts_1
storkc_pp(:,:,iel)=kcx*kx+ky*ky+kz*kz
END DO elements_1
!----- build the diagonal preconditioner -----
ALLOCATE(diag_precon_tmp(ntot,nels_pp)); diag_precon_tmp=zero
elements_1a: DO iel=1,nels_pp
    DO i=1,ndof
        diag_precon_tmp(i,iel)=diag_precon_tmp(i,iel)+storkc_pp(i,i,iel)
    END DO
END DO elements_1a; CALL scatter(diag_pp,diag_precon_tmp)
DEALLOCATE(diag_precon_tmp)
!----- read in fixed freedoms and assign to equations -----
IF(fixed Freedoms>0) THEN
    ALLOCATE(node(fixed Freedoms),no_pp_temp(fixed Freedoms),
             val_f(fixed Freedoms),no(fixed Freedoms),sense(fixed Freedoms)) &
    node=0; no=0; no_pp_temp=0; sense=0; val_f=zero
    CALL read_fixed(argv,numpe,node,sense,val_f)
    CALL find_no2(g_g_pp,g_num_pp,node,sense,no)
    CALL reindex(ieq_start,no,no_pp_temp,fixed Freedoms_pp,
                 fixed Freedoms_start,neq_pp)
    ALLOCATE(no_f_pp(fixed Freedoms_pp),store_pp(fixed Freedoms_pp))
    no_f_pp=0; store_pp=zero; no_f_pp=no_pp_temp(1:fixed Freedoms_pp)
    DEALLOCATE(node,no,sense,no_pp_temp)
END IF
IF(fixed Freedoms==0) fixed Freedoms_pp=0; IF(nr>0) DEALLOCATE(rest)
!----- read in loaded freedoms and get starting r_pp -----
IF.loaded Freedoms>0) THEN
    ALLOCATE(node(loadered Freedoms),val(nodof,loadered Freedoms),
             no_pp_temp(loadered Freedoms)); val=zero; node=0
    CALL read_loads(argv,numpe,node,val)
    CALL reindex(ieq_start,node,no_pp_temp,loaded Freedoms_pp,
                 loadered Freedoms_start,neq_pp); ALLOCATE(no_pp(loadered Freedoms_pp))
    no_pp=no_pp_temp(1:loaded Freedoms_pp); DEALLOCATE(no_pp_temp)
    DO i = 1, loaded Freedoms_pp
        r_pp(no_pp(i)-ieq_start+1) = val(1,loaded Freedoms_start+i-1)
    END DO; q=SUM_P(r_pp); DEALLOCATE(node,val)
END IF
!----- invert preconditioner -----
IF(fixed Freedoms_pp>0) THEN
    DO i=1,fixed Freedoms_pp; j=no_f_pp(i)-ieq_start+1
        diag_precon_pp(j)=diag_precon_pp(j)+penalty
        store_pp(i)=diag_precon_pp(j)
    END DO
END IF; diag_precon_pp = 1._iwp/diag_precon_pp
!----- initialise preconditioned conjugate gradient -----
IF(fixed Freedoms_pp>0) THEN
    DO i=1,fixed Freedoms_pp
        j=no_f_pp(i)-ieq_start+1; k=fixed Freedoms_start+i-1

```

```

r_pp(j)=store_pp(i)*val_f(k)
END DO
END IF; d_pp=diag_precon_pp*r_pp; p_pp=d_pp; x_pp=zero
!----- preconditioned conjugate gradient iterations -----
iters=0; timest(3)=elap_time()
iterations: DO
  iters=iters+1; u_pp=zero; pmul_pp=zero; utemp_pp=zero
  CALL gather(p_pp,pmul_pp)
  elements_2 : DO iel = 1, nels_pp
    utemp_pp(:,iel) = MATMUL(storkc_pp(:,:,iel),pmul_pp(:,iel))
  END DO elements_2 ; CALL scatter(u_pp,utemp_pp)
  IF(fixed_freeoms_pp>0) THEN
    DO i=1,fixed_freeoms_pp
      j=no_f_pp(i)-ieq_start+1; u_pp(j)=p_pp(j)*store_pp(i)
    END DO
  END IF
  up=DOT_PRODUCT_P(r_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
  xnew_pp=x_pp+p_pp*alpha; r_pp=r_pp-u_pp*alpha
  d_pp=diag_precon_pp*r_pp; beta=DOT_PRODUCT_P(r_pp,d_pp)/up
  p_pp=d_pp+p_pp*beta; CALL checon_par(xnew_pp,tol,converged,x_pp)
  IF(converged .OR. iters==limit) EXIT
END DO iterations
timest(4)=elap_time()
IF(numpe==it)THEN
  WRITE(11,'(A,I5)') "The number of iterations to convergence was ",iters
  WRITE(11,'(A,E12.4)') "The total load is ",q
  WRITE(11,'(A)')      "The potentials are:"
  WRITE(11,'(A)')      " Freedom          Potential"
  DO i=1,4
    WRITE(11,'(I8,A,E12.4)') nres+i-1, "      ", xnew_pp(is+i-1)
  END DO
  WRITE(11,'(A,F10.4)') "Integration, preconditioning and loading took ",&
    timest(3)-timest(2)
  WRITE(11,'(A,F10.4)') "Time spent in the solver was ",           &
    timest(4)-timest(3)
END IF
!----- output potentials for ParaView -----
IF(numpe==1)THEN; WRITE(ch,'(I6.6)') numpe
  OPEN(12,file=argv(1:nlen)//".ensi.NDPTL"//ch,status='replace',      &
    action='write')
  WRITE(12,'(A)') "Alya Ensight Gold --- Scalar per-node variable file"
  WRITE(12,'(A/A/A)') "part", " 1","coordinates"
END IF
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
ALLOCATE(ptl_pp(nodes_pp*ndim))
ptl_pp=zero; utemp_pp=zero; CALL gather(xnew_pp(1:),utemp_pp)
CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,ndim,nodes_pp,        &
  node_start,node_end,utemp_pp,ptl_pp,1)
CALL dismsh_ensi_p(12,1,nodes_pp,npes,numpe,1,ptl_pp)
IF(numpe==it)
  WRITE(11,'(A,F10.4)') "This analysis took ",elap_time()-timest(1)
IF(numpe==it) CLOSE(11); IF(numpe==1) CLOSE(12); CALL SHUTDOWN()
END PROGRAM p123

```

The closest serial equivalent to this program is Program 7.5. In the test problem generated by p12meshgen, geometrical restrictions are to a quarter of a cuboidal box with

zero potential on all its outer faces. A potential of 100.0 units can be fixed at the centre of the box or a flux of 10.0 units applied there, so either `loaded_freedoms` or `fixed_freedoms` must be set to 1 and the other to 0.

Comparison of Programs 7.5 and 12.3 will show the familiar patterns of analysis type and parallelisation process, respectively. Loops `elements_1` and `elements_2` are equivalent in both versions. An extra loop `elements_1a` is necessary for formation of the preconditioner. In parallel, flow rates are not retrieved at the end of the analysis. The appropriate freedom for output of summary results is identified as `is` and the processor on which it resides as `it`. The first process collects distributed nodal potentials `p1_pp` and writes them to a file so that they can be visually inspected using ParaView.

The problem analysed with the data shown in Figure 12.18 is of a quarter cube of unit size with a flux of 10.0 units applied at the centre. Eight million equations are involved in this analysis. The potential at the centre of the cube is computed to be 3498.0 units. Results are listed as Figure 12.19 and illustrations of increasing pcg iteration counts with problem size, together with speedup for different code sections, are shown in Figure 12.20.

Figure 12.20 shows that in this example, the analysis proceeds so quickly that reading and writing the results (which here is a serial operation) dominates the analysis time. The time spent in the pcg solver reduces almost linearly with increasing processes and then appears to level off at 64 processes. The element integration, preconditioning and loading

```

program
'p123'

iotype    nels    nxe nze nip
'parafem' 8000000 200 200 8

aa      bb      cc      kx      ky      kz
0.005  0.005  0.005  2.0    2.0    2.0

tol      limit
1.0E-5 1000

loaded_freedoms fixed_freedoms
1                  0

```

**Figure 12.18** Data for Program 12.3 example

```

This job ran on 16 processes
There are 8120601 nodes 120601 restrained and 8000000 equations
Time after setup is 48.2350
The number of iterations to convergence was 180
The total load is 0.1000E+02
The potentials are:
Freedom Potential
39801  0.3498E+04
39802  0.3447E+03
39803  0.3193E+03
39804  0.2004E+03
Integration, preconditioning and loading took 9.4406
Time spent in the solver was 52.3833
This analysis took 117.8274

```

**Figure 12.19** Results from Program 12.3 example

Mesh	Processes	Solver(s)	Integration(s)	Analysis(s)
200x200x200	8	89	19	164
	16	53	9	118
	32	26	4.7	88
	64	20	2.8	86

Number of Equations	Iterations to Convergence
125,000	72
1,000,000	121
8,000,000	180

**Figure 12.20** Performance statistics: Program 12.3 (Cray XE6)

sections of the program require very little inter-process communication and therefore the scalability in those sections is better (see column headed Integration). However, overall, the time for the full analysis only reduces by half when using eight times as many processes.

The issue here is that the external files hold data in ASCII format. On reading, the program has to convert each ASCII character into the binary form used by computers to execute instructions. This is a slow process. Fortunately, the bottleneck can be removed using a BINARY format. BINARY read and write operations proceed much more quickly, as will be illustrated for Program 12.4. Note that this issue did not arise in Program 12.2 as the proportion of time reading and writing is very small in comparison to the other parts of the program.

## Program 12.4 Three-dimensional transient heat conduction–implicit analysis in time. Compare Program 8.5

```
PROGRAM p124
!-----
!      program 12.4 three dimensional transient analysis of heat conduction
!      equation using 8-node hexahedral elements; parallel pcg version
!      implicit; integration in time using 'theta' method
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE geometry; USE new_library; IMPLICIT NONE
!neq,ntot are now global variables - not declared
INTEGER, PARAMETER::ndim=3,nodof=1,nprops=5
INTEGER::nod,nn,nr,nip,i,j,k,l,itors,limit,iel,nstep,npri,nres,it,prog, &
nlen,node_end,node_start,nodes_pp,loaded_freedoms,fixed_freedoms,is, &
fixed_freedoms_pp,fixed_freedoms_start,loaded_freedoms_pp,np_types, &
loaded_freedoms_start,nels,ndof,npes_pp,meshgen,partitioner,tz
REAL(iwp)::kx,ky,kz,det,theta,dtim,real_time,tol,alpha,beta,up,big,q, &
rho,cp,val0
REAL(iwp),PARAMETER::zero=0.0_iwp,penalty=1.e20_iwp,t0=0.0_iwp
CHARACTER(LEN=15)::element; CHARACTER(LEN=50)::argv,fname
CHARACTER(LEN=6)::ch; LOGICAL::converged=.false.
REAL(iwp),ALLOCATABLE::loads_pp(:),u_pp(:),p_pp(:),points(:, :, :),kay(:, :, :), &
fun(:, :),jac(:, :, :),der(:, :, :),deriv(:, :, :),weights(:, :),d_pp(:, :),col(:, :, :), &
```

```

kc(:,:,),pm(:,:),funny(:,:),storka_pp(:,:,:),row(:,:),prop(:,:),
storkb_pp(:,:,:),x_pp(:,xnew_pp(:,pmul_pp(:,:)),utemp_pp(:,:),
diag_precon_pp(:,diag_precon_tmp(:,:),g_coord_pp(:,:)),timest(:),
ttr_pp(:,eld_pp(:,:),val(:,:),val_f(:,store_pp(:,r_pp(:),
kcx(:,:,kcy(:,:,kcz(:,:,eld(:)
INTEGER,ALLOCATABLE::rest(:,:),g_num_pp(:,:),g_g_pp(:,:),no(:),
no_pp(:),no_f_pp(:),no_pp_temp(:),sense(:),node(:),etype_pp(:)
!-----input and initialisation-----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p124(argv,numpe,dtim,element,fixed_freedoms,limit,
loaded_freedoms,meshgen,nels,nip,nn,nod,npri,nr,nres,nstep,
partitioner,theta,tol,np_types,val0)
CALL calc_nels_pp(argv,nels,npes,numpe,partitioner,nels_pp)
ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod,nels_pp),g_coord_pp(nod,ndim,nels_pp),
etype_pp(nels_pp),prop(nprops,np_types))
g_num_pp=0; g_coord_pp=zero; etype_pp=0; prop=zero
IF (nr>0) THEN; ALLOCATE(rest(nr,nodof+1)); rest=0; END IF
CALL read_elements(argv,iel_start,nn,npes,numpe,etype_pp,g_num_pp)
IF(meshgen==2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
IF (nr>0) CALL read_rest(argv,numpe,rest)
CALL read_material(argv,prop,numpe,npes)
ALLOCATE (points(nip,ndim),weights(nip),kay(ndim,ndim),fun(nod),
jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),pm(ntot,ntot),
kc(ntot,ntot),funny(1,nod),g_g_pp(ntot,nels_pp),
storka_pp(ntot,ntot,nels_pp),eld(ntot),col(ntot,1),row(1,ntot),
utemp_pp(ntot,nels_pp),storkb_pp(ntot,ntot,nels_pp),
pmul_pp(ntot,nels_pp),kcx(ntot,ntot),kcy(ntot,ntot),kcz(ntot,ntot))
!----- find the steering array and equations per process -----
timest(2)=elap_time(); g_g_pp=0; neq=0
IF(nr>0) THEN; CALL rearrange_2(rest)
elements_1: DO iel = 1, nels_pp
    CALL find_g4(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_1
ELSE
    g_g_pp=g_num_pp !When nr = 0, g_num_pp and g_g_pp are identical
END IF
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN;it=numpe;is=i;END IF;END DO
IF(numpe==it)THEN
    OPEN(11,FILE=argv(1:nlen)//'.res',STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I5,A)')"This job ran on ", npes, " processes"
    WRITE(11,'(A,3(I7,A))')"There are ",nn," nodes",nr,
        " restrained and ",neq," equations"
    WRITE(11,'(A,F10.4)')"Time after setup is ",elap_time()-timest(1)
END IF
ALLOCATE(loads_pp(neq_pp),diag_precon_pp(neq_pp),u_pp(neq_pp),
d_pp(neq_pp),p_pp(neq_pp),x_pp(neq_pp),xnew_pp(neq_pp),r_pp(neq_pp))
loads_pp=zero; diag_precon_pp=zero; u_pp=zero; r_pp=zero; d_pp=zero
p_pp=zero; x_pp=zero; xnew_pp=zero
!----- element stiffness integration and storage -----
CALL sample(element,points,weights); storka_pp=zero; storkb_pp=zero
elements_3: DO iel=1,nels_pp
    kay=zero; kc=zero; pm=zero; kay(1,1)=prop(1,etype_pp(iel))
    kay(2,2)=prop(2,etype_pp(iel)); kay(3,3)=prop(3,etype_pp(iel))

```

```

rho=prop(4,etype_pp(iel)); cp=prop(5,etype_pp(iel))
gauss_pts: DO i=1,nip
    CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
    funny(1,:)=fun(:); jac=MATMUL(der,g_coord_pp(:,:,iel))
    det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
    kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
    pm=pm+MATMUL(TRANSPOSE(funny),funny)*det*weights(i)*rho*cp
END DO gauss_pts
storka_pp(:,:,iel)=pm+kc*theta*dtim
storkb_pp(:,:,iel)=pm-kc*(1._iwp-theta)*dtim
END DO elements_3
!----- build the diagonal preconditioner -----
ALLOCATE(diag_precon_tmp(ntot,nels_pp)); diag_precon_tmp = zero
elements_4: DO iel = 1,nels_pp
    DO k=1,ntot
        diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)+storka_pp(k,k,iel)
    END DO
END DO elements_4; CALL scatter(diag_precon_pp,diag_precon_tmp)
DEALLOCATE(diag_precon_tmp)
!----- read in fixed freedoms and assign to equations -----
IF(fixed Freedoms > 0) THEN
    ALLOCATE(node(fixed Freedoms),no_pp_temp(fixed Freedoms),
             &
             no(fixed Freedoms),sense(fixed Freedoms),val_f(fixed Freedoms))
    node=0; no=0; no_pp_temp=0; sense=0; val_f = zero
    CALL read_fixed(argv,numpe,node,sense,val_f)
    CALL find_no2(g_g_pp,g_num_pp,node,sense,no)
    CALL reindex(ieq_start,no,no_pp_temp,fixed Freedoms_pp,
                 &
                 fixed Freedoms_start,neq_pp)
    ALLOCATE(no_f_pp(fixed Freedoms_pp),store_pp(fixed Freedoms_pp))
    no_f_pp=0; store_pp=zero; no_f_pp=no_pp_temp(1:fixed Freedoms_pp)
    DEALLOCATE(node,no,sense,no_pp_temp)
END IF
IF(fixed Freedoms==0) fixed Freedoms_pp=0
!----- invert preconditioner -----
IF(fixed Freedoms_pp > 0) THEN
    DO i=1,fixed Freedoms_pp; l=no_f_pp(i)-ieq_start+1
        diag_precon_pp(l)=diag_precon_pp(l)+penalty
        store_pp(i)=diag_precon_pp(l)
    END DO
END IF; diag_precon_pp=1._iwp/diag_precon_pp
!----- read in loaded nodes and get starting r_pp -----
IF.loaded Freedoms>0) THEN
    ALLOCATE(node(loader Freedoms),val(nodof,loader Freedoms),
             &
             no_pp_temp(loader Freedoms)); val=zero; node=0; no_pp_temp=0
    CALL read_loads(argv,numpe,node,val)
    CALL reindex(ieq_start,node,no_pp_temp,loader Freedoms_pp,
                 &
                 loader Freedoms_start,neq_pp); ALLOCATE(no_pp(loader Freedoms_pp))
    no_pp=0; no_pp=no_pp_temp(1:loader Freedoms_pp)
    DEALLOCATE(no_pp_temp,node)
END IF
!----- start time stepping loop -----
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
ALLOCATE(ttr_pp(nodes_pp),eld_pp(ntot,nels_pp))
ttr_pp=zero; eld_pp=zero
IF(numpe==it)
    WRITE(11,'(A)') " Time Temperature Iterations "
    IF(numpe==1) THEN
        OPEN(13, file=argv(1:nlen)//'.npp', status='replace', action='write')
    END IF
    !----- solve -----
    !----- output -----
    !----- end time step -----
END IF
!----- end time stepping loop -----

```

```

      WRITE(13,*) nn; WRITE(13,*) nstep/npri; WRITE(13,*) npes
END IF
timesteps: DO j=1,nstep
    real_time=j*dtim; timest(3)=elap_time(); loads_pp=zero
!---- apply loads (sources and/or sinks) supplied as a boundary value ----
    IF.loaded_freedoms_pp>0) THEN
        DO i=1,loaded_freedoms_pp; k=no_pp(i)-ieq_start+1
            loads_pp(k)=val.loaded_freedoms_start+i-1,1)*dtim
        END DO; q=q+SUM_P(loads_pp)
    END IF
!- compute RHS of time stepping equation, using storkb_pp, add to loads --
    u_pp=zero; pmul_pp=zero; utemp_pp=zero
    IF(j/=1) THEN
        CALL gather(xnew_pp,pmul_pp)
        elements_2a: DO iel=1,nels_pp
            utemp_pp(:,iel)=MATMUL(storkb_pp(:,:,iel),pmul_pp(:,iel))
        END DO elements_2a; CALL scatter(u_pp,utemp_pp)
        IF(fixed_freedoms_pp>0) THEN
            DO i=1,fixed_freedoms_pp; l=no_f_pp(i)-ieq_start+1
                k=fixed_freedoms_start+i-1; u_pp(l)=store_pp(i)*val_f(k)
            END DO
        END IF; loads_pp=loads_pp+u_pp
    ELSE
!----- set initial temperature -----
        x_pp=val0; IF(numpe==it) WRITE(11,'(2e12.4)') 0.0_iwp, x_pp(is)
        IF(fixed_freedoms_pp>0) THEN
            DO i=1,fixed_freedoms_pp; l=no_f_pp(i)-ieq_start+1
                k=fixed_freedoms_start+i-1; x_pp(l)=val_f(k)
            END DO
        END IF
        CALL gather(x_pp,pmul_pp)
        elements_2c: DO iel=1,nels_pp
            utemp_pp(:,iel)=MATMUL(storka_pp(:,:,iel),pmul_pp(:,iel))
        END DO elements_2c; CALL scatter(u_pp,utemp_pp)
        loads_pp=loads_pp+u_pp; tz=0
!----- output "results" at t=0 -----
        IF(numpe==1)THEN; WRITE(ch,'(I6.6)') tz
            OPEN(12,file=argv(1:nlen)//".ensi.NDTTR-"/ch,status='replace', &
            action='write')
            WRITE(12,'(A)') &
                "Alya Ensight Gold --- Scalar per-node variable file"
            WRITE(12,'(A/A/A)') "part", " 1", "coordinates"
        END IF
        eld_pp=zero; ttr_pp=zero; CALL gather(x_pp(1:),eld_pp)
        CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,nodof,nodes_pp, &
            node_start,node_end,eld_pp,ttr_pp,1)
        CALL dismsh_ensi_p(12,1,nodes_pp,npes,numpe,1,ttr_pp)
    !   CALL dismsh_pb(12,1,nodes_pp,npes,numpe,1,ttr_pp)
    END IF
!---- when x=0. p and r are just loads but in general p=r=loads-A*x -----
    r_pp=zero; pmul_pp=zero; utemp_pp=zero; x_pp=zero
    CALL gather(x_pp,pmul_pp)
    elements_2b: DO iel=1,nels_pp
        utemp_pp(:,iel)=MATMUL(storka_pp(:,:,iel),pmul_pp(:,iel))
    END DO elements_2b; CALL scatter(r_pp,utemp_pp)
    IF(fixed_freedoms_pp>0) THEN
        DO i=1,fixed_freedoms_pp; l=no_f_pp(i)-ieq_start+1
            k=fixed_freedoms_start+i-1; r_pp(l)=store_pp(i)*val_f(k)
    END IF

```

```

        END DO
    END IF
    r_pp=loads_pp-r_pp; d_pp=diag_precon_pp*r_pp; p_pp=d_pp; iters=0
!----- solve simultaneous equations by pcg -----
    iterations: DO
        iters=iters+1; u_pp=zero; pmul_pp=zero; utemp_pp=zero
        CALL gather(p_pp,pmul_pp)
        elements_6: DO iel=1,nels_pp
            utemp_pp(:,iel)=MATMUL(storka_pp(:,:,iel),pmul_pp(:,iel))
        END DO elements_6; CALL scatter(u_pp,utemp_pp)
        IF(fixed_freedoms_pp>0) THEN; DO i=1,fixed_freedoms_pp
            l=no_f_pp(i)-ieq_start+1; u_pp(l)=p_pp(l)*store_pp(i)
        END DO; END IF
        up=DOT_PRODUCT_P(r_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
        xnew_pp=x_pp+p_pp*alpha; r_pp=r_pp-u_pp*alpha
        d_pp=diag_precon_pp*r_pp; beta=DOT_PRODUCT_P(r_pp,d_pp)/up
        p_pp=d_pp+p_pp*beta; CALL checon_par(xnew_pp,tol,converged,x_pp)
        IF(converged.OR.iters==limit)EXIT
    END DO iterations; timest(4)=timest(4)+(elap_time()-timest(3))
    IF(j/npri*npri==j)THEN; timest(5)=elap_time()
    IF(numpe==1)THEN; WRITE(ch,'(I6.6)') j
        OPEN(12,file=argv(1:nlen)//".ensi.NDTTR-//ch,status='replace', &
              action='write')
        WRITE(12,'(A)') &
            "Alya Ensight Gold --- Scalar per-node variable file"
        WRITE(12,'(A/A/A)') "part", " 1","coordinates"
    END IF; eld_pp=zero; ttr_pp=zero; CALL gather(xnew_pp(1:),eld_pp)
    CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,nodof,nodes_pp, &
                      node_start,node_end,eld_pp,ttr_pp,1)
    CALL dismsh_ensi_p(12,1,nodes_pp,npes,numpe,1,ttr_pp)
!    CALL dismsh_pb(12,1,nodes_pp,npes,numpe,1,ttr_pp)
    IF(numpe==1) CLOSE(12)
    IF(numpe==it) WRITE(11,'(2E12.4,I10)') real_time,xnew_pp(is),iters
    timest(6)=timest(6)+(elap_time()-timest(5))
    END IF
END DO timesteps
IF(numpe==it) THEN
    WRITE(11,'(A,F10.4)') "The solution phase took ",timest(4)
    WRITE(11,'(A,F10.4)') "Writing the output took ",timest(6)
    WRITE(11,'(A,F10.4)') "This analysis took ",elap_time()-timest(1)
    CLOSE(11)
END IF; CALL SHUTDOWN()
END PROGRAM p124

```

The closest serial equivalent is Program 8.5. This program solves the transient heat conduction equation. The conductivities  $k_{cx}$ ,  $k_{cy}$  and  $k_{cz}$  used in the previous program become ‘thermal conductivities’ and a further two material parameters are required to complete the governing equations, the specific heat capacity  $cp$  and the mass density  $\rho$ . Looking at the `elements_1` loop, when  $cp$  and  $\rho$  have a value of one, Program 12.4 solves the same ‘transient flow’ problem as Program 12.5.

As in the previous program, the geometry of the test problem is restricted to a symmetrical quarter cuboidal box mesh. Here it has a temperature of zero on all external faces. An initial condition of uniform temperature  $val0$  is specified everywhere else and the rate of cooling at the centre of the box is monitored.

Loops `elements_1` and `element_2` with its embedded `gauss_pts` can be traced clearly between the serial and parallelised versions, as can loops `elements_3` and `elements_4`.

This program supports the definition of `np_types` different materials. The material properties, stored in a file, are read into the array `prop` by the first process and broadcast to the other processes using `read_material`. The material type for each element is stored in the distributed array `etype_pp`. Subroutine `read_g_num_pp` used in the previous programs in this chapter is replaced by `read_elements`, which reads and distributes both `g_num_pp` and `etype_pp`.

Nodal temperatures may be output in ASCII format using `dismsh_ensi_p` or in BINARY format using `dismsh_pb`. The BINARY output is written, unformatted, in `npes` separate blocks (each block is formed as the first process alternates between collecting and writing data for each of the other processes). To later read the file, the number of blocks in the BINARY file needs to be known. This information is written to the `pbb` file.

The test problem analysed, as shown in Figure 12.2, is for a quarter cube of unit size with an initial temperature of 100.0 units everywhere except at the external boundaries. Data are listed as Figure 12.21 with results as Figure 12.22 and performance statistics as Figure 12.23.

Program 12.4 has also been used to study heat flow in composites (Evans *et al.*, 2013). Figure 12.24 shows an X-ray tomography scan of a woven silicon–carbide composite being evaluated for possible use in fusion reactors. Figure 12.25 shows a finite element mesh, comprising 125 million tetrahedra, that was generated from the image using specialist software from Simpleware Ltd (see <http://www.simpleware.com>). Figure 12.26 provides a close-up view of the top surface of the mesh. 3D imaging techniques typically produce ‘voxelised’ data. The meshing tool used here turns the voxels into a smoothed mesh (essential for thermal problems) using a volume-preserving algorithm.

Figure 12.27 illustrates how the presence of pores in the material, that have formed as a byproduct of the manufacturing process, influences the temperature profile at one particular instant in time.

A typical performance plot for this problem, Figure 12.28, shows that for models of this size, Program 12.4 scales well on up to 16,000 processes. Early in the development

```

program
'p124'

iotype    nels      nxe nze nip
'parafem' 1000000 100 100 8

aa      bb      cc      kx      ky      kz      rho cp
0.01  0.01  0.01  1.0   1.0   1.0   1.0  1.0

dtim   nsteps theta
0.001 1000    0.5

npri tol      limit val0
100  0.000001 100    100.0

np_types loaded_freedoms fixed_freedoms
1                  0

```

**Figure 12.21** Data for Program 12.4 example

```

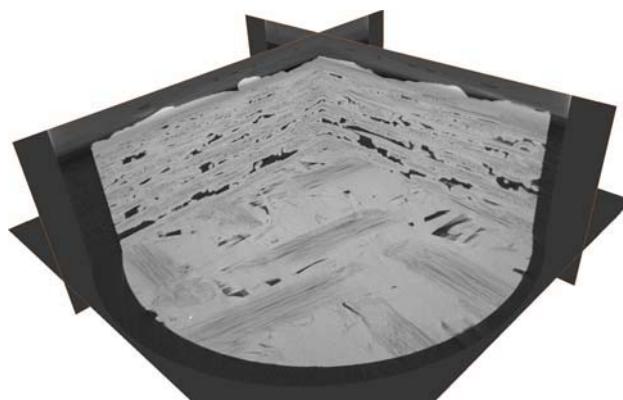
This job ran on    32    processes
There are 1030301 nodes 30301 restrained and 1000000 equations
Time after setup is      6.0724
      Time          Temperature   Iterations
0.0000E+00  0.1000E+03
0.1000E+00  0.8595E+02      15
0.2000E+00  0.4639E+02      11
0.3000E+00  0.2251E+02      10
0.4000E+00  0.1076E+02       9
0.5000E+00  0.5134E+01       7
0.6000E+00  0.2449E+01       5
0.7000E+00  0.1168E+01       5
0.8000E+00  0.5573E+00       4
0.9000E+00  0.2658E+00       4
0.1000E+01  0.1268E+00       4
The solution phase took   184.6235
Writing the output took   10.1006
This analysis took   201.4486

```

**Figure 12.22** Results from Program 12.4 example

Mesh	No of Processes	Analysis Time (secs)
100x100x100	16	385
	32	201
	64	108

**Figure 12.23** Performance statistics: Program 12.4 (Cray XE6)



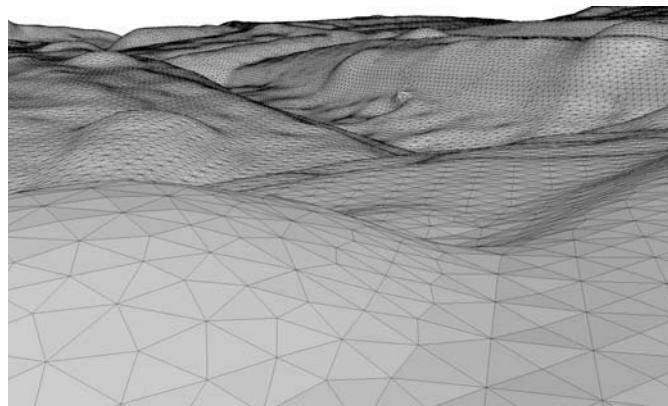
**Figure 12.24** X-ray tomography scan of woven composite

of the program, benchmarking identified two performance bottlenecks, one in the routine `find_no` and the other in writing the results. The sequential `find_no`, which searches for equations, was replaced by a new parallel routine `find_no2`.

Using 16,000 processes, the time to solve 125 million equations at each time step is around 10 s. It takes a comparatively lengthy 50 s per time step to write the results (in ASCII format) using `dismsh_ensi_p`. In contrast, the BINARY routine `dismsh_pb`



**Figure 12.25** Mesh of woven composite with 125 million elements

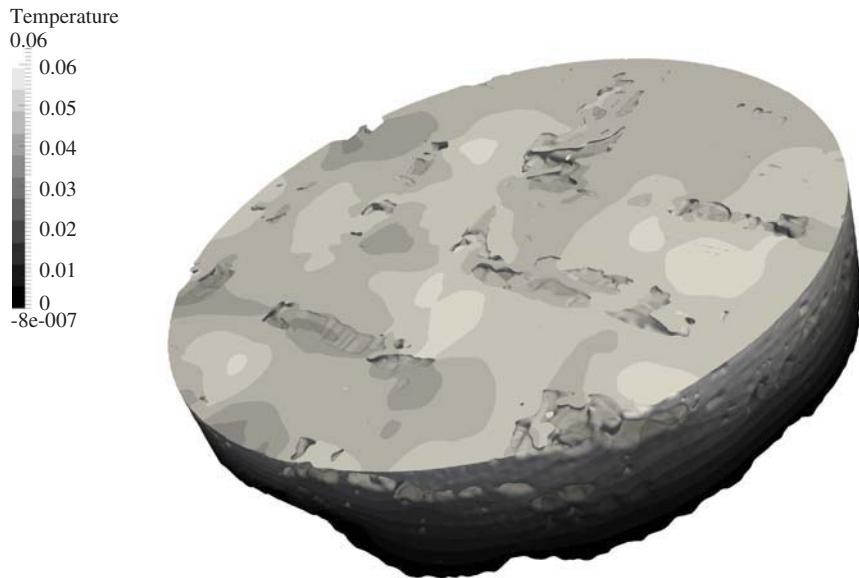


**Figure 12.26** Close-up of the surface of the mesh

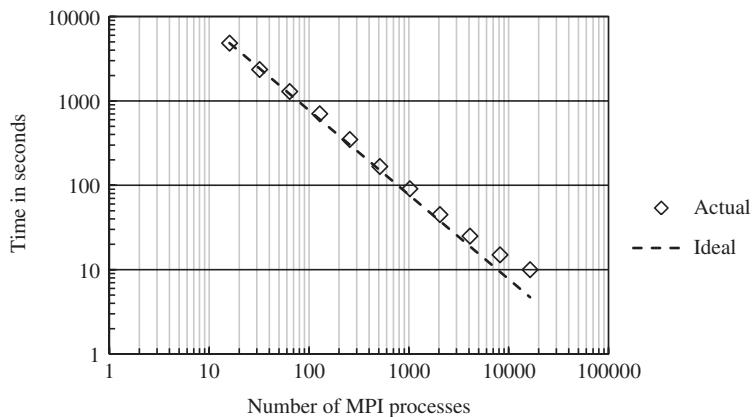
takes less than 0.5 s. Furthermore, if output is only required every 10 or 100 time steps (which is typical), then writing in `BINARY` format becomes insignificant compared with the computation time, even on systems with tens of thousands of processes for problems with more than a hundred million unknowns.

There are various advantages to ‘image-based modelling’, including the ability to model materials with complex architectures and capture the fabrication defects (flaws, cracks, porosity) that would be simplified away using traditional computer-aided design (CAD) techniques. However, the consequence of including this detail is that the models can be very large and at the time of writing, X-ray tomography systems can produce 3D images with up to  $8000 \times 8000 \times 8000$  (512 billion) voxels.

Image-based modelling originated in the biosciences, where creating finite element meshes of geometrically complicated biological structures is very difficult. A typical modelling workflow is carried out as follows. First, it is necessary to capture a 3D digital



**Figure 12.27** Typical plot of temperature



**Figure 12.28** Performance statistics for 125 million equations: Program 12.4 (Cray XE6)

image. Any appropriate imaging technique can be used, including magnetic resonance imaging or scanning electron microscopy. The next step is to ‘segment’ the image into distinct structures (bone, muscle, fat, air). These structures are then meshed using either simple algorithms that convert the voxels into hexahedral elements or more complicated procedures that produce tetrahedral meshes with smooth surfaces. Once the mesh has been generated, the user sets boundary conditions and material properties as usual, then runs the analysis. For a biological example, see Brassey *et al.* (2013). In their paper, they use image-based modelling to investigate the role of curvature in the mechanics of vertebrate long bones.

## Program 12.5 Three-dimensional transient flow-explicit analysis in time. Compare Program 8.6

```

PROGRAM p125
!-----
! Program 12.5 conduction equation on a 3-d box volume using 8-node
! hexahedral elements and a simple explicit algorithm : parallel version
! write on processor it at freedom nres
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE geometry; USE new_library; IMPLICIT NONE
! neq,ntot are now global variables - not declared
INTEGER,PARAMETER::nodof=1,ndim=3
INTEGER::nels(ndof,npes_pp,nn,nr,nip,nod,i,j,k,iel,nstep,nlen,
  npri,nres,it,is,meshgen,partitioner,loaded_nodes=0,fixed_freeoms,
  nodes_pp,node_start,node_end
REAL(iwp)::kx,ky,kz,det,dtim,val0,real_time
REAL(iwp),PARAMETER::zero=0.0_iwp; CHARACTER(LEN=6)::ch
CHARACTER(LEN=15)::element; CHARACTER(LEN=50)::argv
!----- dynamic arrays -----
REAL(iwp),ALLOCATABLE::loads_pp(:,points(:, :, :),kay(:, :, :),jac(:, :, :),
  der(:, :, :),deriv(:, :, :),weights(:, :, :),kc(:, :, :),funny(:, :, :),globma_pp(:, :, :),
  fun(:, :, :),store_pm_pp(:, :, :, :),newlo_pp(:, :, :),mass(:, :, :),globma_tmp(:, :, :),
  pmul_pp(:, :, :),utemp_pp(:, :, :),g_coord_pp(:, :, :, :),timest(:, :, :),pm(:, :, :),
  ptl_pp(:, :, :)
INTEGER,ALLOCATABLE::rest(:, :, :),g_num_pp(:, :, :),g_g_pp(:, :, :)
!----- input and initialisation -----
ALLOCATE(timest(25)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p125(argv,numpe,dtim,element,fixed_freeoms,kx,ky,kz,
  loaded_nodes,meshgen,nels,nip,nn,nod,npri,nr,nres,nstep,partitioner,
  val0)
CALL calc_nels_pp(argv,nels,npes,numpe,partitioner,nels_pp)
ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod,nels_pp),g_coord_pp(nod,ndim,nels_pp),
  rest(nr,nodof+1)); g_num_pp=0; g_coord_pp=zero; rest=0
CALL read_g_num_pp(argv,iel_start,nn,npes,numpe,g_num_pp)
IF(meshgen==2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
CALL read_rest(argv,numpe,rest); timest(2)=elap_time()
ALLOCATE (points(nip,ndim),weights(nip),kay(ndim,ndim),jac(ndim,ndim),
  der(ndim,nod),deriv(ndim,nod),kc(ntot,ntot),funny(1,nod),
  g_g_pp(ntot,nels_pp),store_pm_pp(ntot,ntot,nels_pp),
  mass(ntot),fun(nod),pm(ntot,ntot),globma_tmp(ntot,nels_pp),
  pmul_pp(ntot,nels_pp),utemp_pp(ntot,nels_pp))
!----- find the steering array and equations per process -----
CALL rearrange_2(rest); g_g_pp = 0; neq=0
elements_0: DO iel = 1, nels_pp
  CALL find_g4(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_0
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN;it=numpe;is=i;END IF;END DO
IF(numpe==it)THEN

```

```

OPEN(11,FILE=argv(1:nlen)//".res",STATUS='REPLACE',ACTION='WRITE')
WRITE(11,'(A,I5,A)')"This job ran on ",npes," processes"
WRITE(11,'(A,3(I12,A))')"There are ",nn," nodes",nr," restrained and",&
    neq," equations"
WRITE(11,'(A,F10.4)') "Time to read input is:",timest(2)-timest(1)
WRITE(11,'(A,F10.4)') "Time after setup is:",elap_time()-timest(1)
END IF
ALLOCATE(loads_pp(neq_pp),newlo_pp(neq_pp),globma_pp(neq_pp))
loads_pp=zero; newlo_pp=zero; globma_pp=zero; globma_tmp=zero
!----- loop the elements for integration and invert mass -----
timest(3)=elap_time(); CALL sample(element,points,weights)
kay=zero; kay(1,1)=kx; kay(2,2)=ky; kay(3,3)=kz
elements_1: DO iel=1,nels_pp
    kc=zero; pm=zero
    gauss_pts: DO i=1,nip
        CALL shape_der(dер,points,i); CALL shape_fun(fun,points,i)
        funny(1,:)=fun(:,); jac=MATMUL(der,g_coord_pp(:,:,iel))
        det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
        kc=kc+MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i)
        pm=pm+MATMUL(TRANSPOSE(funny),funny)*det*weights(i)
    END DO gauss_pts
    DO i=1,ntot; mass(i)=sum(pm(i,:)); END DO
    pm=zero; DO i=1,ntot; pm(i,i)=mass(i); END DO
    store_pm_pp(:,:,iel)=pm-kc*dtim
    DO i=1,ntot; globma_tmp(i,iel)=globma_tmp(i,iel)+mass(i); END DO
END DO elements_1
IF(numpe==it) THEN; WRITE(11,'(A,F10.4)') &
    "Time for element integration is :",elap_time()-timest(3); END IF
CALL scatter(globma_pp,globma_tmp); globma_pp = 1._iwp/globma_pp
loads_pp=val0; DEALLOCATE(globma_tmp); timest(4)=elap_time()
!----- time stepping recursion -----
IF(numpe==it)THEN
    WRITE(11,'(A)')" Time Pressure"
    WRITE(11,'(2E12.4)') 0.0_iwp, loads_pp(is)
END IF
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
ALLOCATE(ptl_pp(nodes_pp*ndim))
timesteps: DO j=1,nstep
    real_time = j*dtim
!----- go round the elements -----
    utemp_pp=zero; pmul_pp=zero; CALL gather(loads_pp,pmul_pp)
    elements_2: DO iel=1,nels_pp
        pm = store_pm_pp(:,:,iel)
        utemp_pp(:,iel) = utemp_pp(:,iel)+MATMUL(pm,pmul_pp(:,:,iel))
    END DO elements_2; CALL scatter(newlo_pp,utemp_pp)
    loads_pp=newlo_pp*globma_pp; newlo_pp=zero
    IF(j/npri*npri==j) THEN
        IF(numpe==it) WRITE(11,'(2E12.4)')real_time,loads_pp(is)
!----- output pressures for ParaView -----
        IF(numpe==1)THEN; WRITE(ch,'(I6.6)') j
            OPEN(12,file=argv(1:nlen)//".ensi.NDPRE-//ch,status='replace', &
                action='write')
            WRITE(12,'(A)')"Alya Ensight Gold --- Scalar per-node variable file"
            WRITE(12,'(A/A/A)') "part", " 1", "coordinates"
        END IF
        pt1_pp=zero; utemp_pp=zero; CALL gather(loads_pp(1:),utemp_pp)
        CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,ndim,nodes_pp, &
            node_start,node_end,utemp_pp,pt1_pp,1)
    END IF

```

```

CALL dismsh_ensi_p(12,1,nodes_pp,npes,numpe,1,pt1_pp); CLOSE(12)
END IF
END DO timesteps; timest(5)=elap_time()
IF(numpe==it) THEN
  WRITE(11,'(A,F10.4)')"Time stepping recursion took :",
    timest(5)-timest(4)
  WRITE(11,'(A,F10.4)')"This analysis took :",elap_time()-timest(1)
END IF; CALL SHUTDOWN()
END PROGRAM p125

```

As is often the case in parallel programs, an extra loop `elements_0` is used to organise the distribution of the freedoms via steering vector `g`. Thereafter, loop `elements_1` with its embedded `gauss_pts` carries over from serial to parallel versions, as does loop `elements_2`. In parallel, results at the appropriate freedom, `is`, are printed from processor `it`. Data are listed as Figure 12.29 with results as Figure 12.30 and performance statistics as Figure 12.31. The problem solved is essentially the same as the test problem used for Program 12.4. Therefore, the results and performance of the implicit and explicit solution strategies can be compared.

```

program
'p125'

iotype     nels      nxe   nze   nip
'parafem'  1000000  100   100   8

aa      bb      cc      kx      ky      kz
0.01  0.01  0.01  1.0   1.0   1.0

dtim     nsteps
0.00005 20000

npri    val0
2000   100.0

```

**Figure 12.29** Data for Program 12.5 example

```

This job ran on 32 processes
There are 1030301 nodes 30301 restrained and 1000000 equations
Time to read input is: 4.0594
Time after setup is: 4.5543
Time for element integration is : 0.3100
  Time          Pressure
  0.0000E+00  0.1000E+03
  0.1000E+00  0.8554E+02
  0.2000E+00  0.4605E+02
  0.3000E+00  0.2233E+02
  0.4000E+00  0.1067E+02
  0.5000E+00  0.5091E+01
  0.6000E+00  0.2427E+01
  0.7000E+00  0.1157E+01
  0.8000E+00  0.5516E+00
  0.9000E+00  0.2628E+00
  0.1000E+01  0.1252E+00
Time stepping recursion took : 383.4017
This analysis took : 388.2760

```

**Figure 12.30** Results from Program 12.5 example

Mesh	No of MPI Processes	Analysis Time (secs)
100x100x100	16	564
	32	388
	64	277

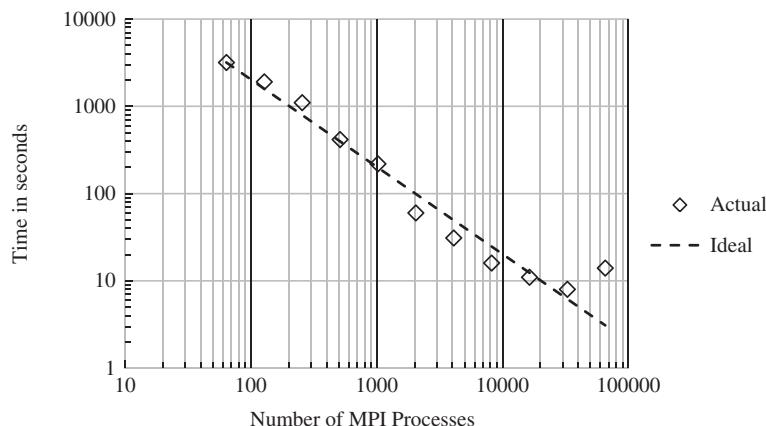
**Figure 12.31** Performance statistics: Program 12.5 (SGI Intel Cluster)**Figure 12.32** Performance statistics for 125 million equations: Program 12.5 (Cray XE6)

Figure 12.32 shows the scalability of the time-stepping recursion section of code on a Cray XE6 system. This problem has 125 million equations and times are recorded for 1000 steps. The graph shows that there is no benefit in using more than 32,000 processes. As discussed for Program 12.4, reading and writing becomes a bottleneck when running large problems on large systems and the use of binary files is essential. That said, the relative proportion of time spent reading and writing diminishes as the number of steps carried out before the next write is increased. On 32,000 processes, over 5 million steps could be performed in a 12-hour overnight run.

## Program 12.6 Three-dimensional steady-state Navier–Stokes analysis. Compare Program 9.2

```

PROGRAM p126
!-----
! Program 12.6 Steady state 3-d Navier-Stokes equation using 20-node
! velocity hexahedral elements coupled to 8-node pressure
! hexahedral elements : u-p-v-w order element by element
! solution using BiCGSTAB(L) : parallel version
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input;
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE steering; USE fluid; USE new_library
IMPLICIT NONE
!neq,ntot are now global variables - not declared

```

```

INTEGER,PARAMETER::nodof=4,nod=20,nodf=8,ndim=3
INTEGER::nn,nip,cj_tot,i,j,k,l,iel,ell,limit,fixed_freedoms,iters,
cjters,cjits,nr,n_t,fixed_freedoms_pp,nres,is,it,nlen,neils,ndof,
npes_pp,meshgen,partitioner,node_end,node_start,nodes_pp,
fixed_freedoms_start
REAL(iwp):: visc,rho,rhol,det,ubar,vbar,wbar,tol,cjtol,alpha,beta,
penalty,x0,pp,kappa,gama,omega,norm_r,r0_norm,error
REAL(iwp),PARAMETER::zero=0.0_iwp,one=1.0_iwp
LOGICAL::converged,cj_converged
CHARACTER(LEN=15)::element='hexahedron'; CHARACTER(LEN=50)::argv
!----- dynamic arrays -----
REAL(iwp),ALLOCATABLE::points(:,:),derivf(:,:),fun(:),store_pp(:),
jac(:,:),kay(:,:),der(:,:),deriv(:,:),weights(:),derf(:,:),funf(:),
coordf(:,:),g_coord_pp(:,:,:),c11(:,:),c21(:,:),c12(:,:),val(:),
wvel(:),c23(:,:),c32(:,:),x_pp(:),b_pp(:),r_pp(:,:),temp(:),
funny(:,:),row1(:,:),row2(:,:),uvel(:),vvel(:),funnyf(:,:),rowf(:,:),
storke_pp(:,:,:),diag_pp(:),utemp_pp(:),xold_pp(:),c24(:,:),
c42(:,:),row3(:,:),u_pp(:,:),rt_pp(:),y_pp(:),y1_pp(:),s(:),Gamma(:),
GG(:,:),diag_tmp(:,:),pmul_pp(:,:),timest(:),upvw_pp(:)
INTEGER,ALLOCATABLE::rest(:,:),g_num_pp(:,:),g_g_pp(:,:),no(:,),g_t(:,),
no_pp(:,),no_pp_temp(:,)
!----- input and initialisation -----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p126(argv,numpe,cjits,cjtol,ell,fixed_freedoms,kappa,limit,
meshgen,neils,nip,nn,nr,nres,partitioner,penalty,rho,tol,x0,visc)
CALL calc_neils_pp(argv,neils,npes,numpe,partitioner,neils_pp)
ntot=nod+nodf+nod+nod; n_t=nod*nodof
ALLOCATE(g_num_pp(nod,neils_pp),g_coord_pp(nod,ndim,neils_pp),
rest(nr,nodof+1)); g_num_pp=0; g_coord_pp=zero; rest=0
CALL read_g_num_pp(argv,iel_start,nn,npes,numpe,g_num_pp)
IF(meshgen == 2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
CALL read_rest(argv,numpe,rest); timest(2)=elap_time()
ALLOCATE(points(nip,ndim),derivf(ndim,nodf),pmul_pp(ntot,neils_pp),
jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),
derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),funny(nod,1),s(ell+1),
g_g_pp(ntot,neils_pp),c11(nod,nod),c12(nod,nodf),c21(nodf,nod),
c24(nodf,nod),c42(nod,nodf),c32(nod,nodf),fun(nod),row2(1,nod),
c23(nodf,nod),uvel(nod),vvel(nod),row1(1,nod),funnyf(nodf,1),
rowf(1,nodf),no_pp_temp(fixed_freedoms),wvel(nod),row3(1,nod),
storke_pp(ntot,ntot,neils_pp),g_t(n_t),GG(ell+1,ell+1),
Gamma(ell+1),no(fixed_freedoms),val(fixed_freedoms),weights(nip),
diag_tmp(ntot,neils_pp),utemp_pp(ntot,neils_pp))
!----- find the steering array and equations per process -----
CALL rearrange(rest); g_g_pp=0; neq=0
elements_1: DO iel=1,neils_pp
    CALL find_g3(g_num_pp(:,iel),g_t,rest)
    CALL g_t_g_ns(nod,g_t,g_g_pp(:,iel))
END DO elements_1
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp)
CALL make_ggl(npes_pp,npes,g_g_pp); DEALLOCATE(g_g_pp)
DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN;it=numpe;is=i;END IF;END DO
IF(numpe==it) THEN
    OPEN(11,FILE=argv(1:nlen)//".res",STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I6,A)') "This job ran on ",npes," processes"
    WRITE(11,'(A,3(I12,A))') "There are ",nn," nodes ",nr,
    &

```

```

    " restrained and ", neq, " equations"
WRITE(11,'(A,F10.4)') "Time to read input was:",timest(2)-timest(1)
WRITE(11,'(A,F10.4)') "Time after setup was:",elap_time()-timest(1)
END IF
ALLOCATE(x_pp(neq_pp),rt_pp(neq_pp),r_pp(neq_pp,ell+1),b_pp(neq_pp),
         u_pp(neq_pp,ell+1),diag_pp(neq_pp),xold_pp(neq_pp),y_pp(neq_pp),
         &
         y1_pp(neq_pp),store_pp(neq_pp))
x_pp=zero; rt_pp=zero; r_pp=zero; u_pp=zero; b_pp=zero; diag_pp=zero
xold_pp=zero; y_pp=zero; y1_pp=zero; store_pp=zero
!----- organise fixed equations -----
CALL read_loads_ns(argv,numpe,no,val)
CALL reindex(ieq_start,no,no_pp_temp,fixed_freedoms_pp,
            &
            fixed_freedoms_start,neq_pp); ALLOCATE(no_pp(1:fixed_freedoms_pp))
no_pp = no_pp_temp(1:fixed_freedoms_pp); DEALLOCATE(no_pp_temp)
!----- main iteration loop -----
CALL sample(element,points,weights); uvel=zero; vvel=zero; wvel=zero
kay=zero; iters=0; cj_tot=0; kay(1,1)=visc/rho; kay(2,2)=visc/rho
kay(3,3)=visc/rho; timest(3)=elap_time()
iterations: DO
    iters=iters+1; storke_pp=zero; diag_pp=zero; utemp_pp=zero
    b_pp=zero; pmul_pp=zero; CALL gather(x_pp,utemp_pp)
    CALL gather(xold_pp,pmul_pp)
!----- element stiffness integration -----
elements_2: DO iel=1,nels_pp
    uvel=(utemp_pp(1:nod,iel)+pmul_pp(1:nod,iel))*5_iwp
    DO i=nod+nodf+1,nod+nodf+nod
        vvel(i-nod-nodf)=(utemp_pp(i,iel)+pmul_pp(i,iel))*5_iwp
    END DO
    DO i=nod+nodf+nod+1,ntot
        wvel(i-nod-nodf-nod)=(utemp_pp(i,iel)+pmul_pp(i,iel))*5_iwp
    END DO
    c11=zero; c12=zero; c21=zero; c23=zero; c32=zero; c24=zero; c42=zero
    gauss_points_1: DO i=1,nip
!----- velocity contribution -----
        CALL shape_fun(funny(:,1),points,i)
       ubar=DOT_PRODUCT(funny(:,1),uvel);vbar=DOT_PRODUCT(funny(:,1),vvel)
        wbar=DOT_PRODUCT(funny(:,1),wvel)
        IF(iters==1)THEN; ubar=one; vbar=zero; wbar=zero; END IF
        CALL shape_der(der,points,i); jac=MATMUL(der,g_coord_pp(:,:,iel))
        det=determinant(jac); CALL invert(jac); deriv=MATMUL(jac,der)
        row1(1,:)=deriv(1,:); row2(1,:)=deriv(2,:); row3(1,:)=deriv(3,:)
        c11=c11+
            MATMUL(MATMUL(TRANSPOSE(deriv),kay),deriv)*det*weights(i) +
            MATMUL(funny,row1)*det*weights(i)*ubar +
            MATMUL(funny,row2)*det*weights(i)*vbar +
            MATMUL(funny,row3)*det*weights(i)*wbar
!----- pressure contribution -----
        CALL shape_fun(funnyf(:,1),points,i); CALL shape_der(derf,points,i)
        coordf(1:4,:)=g_coord_pp(1:7:2,:,:,iel)
        coordf(5:8,:)=g_coord_pp(13:19:2,:,:,iel); jac=MATMUL(derf,coordf)
        det=determinant(jac); CALL invert(jac); derivf=MATMUL(jac,derf)
        rowf(1,:)=derivf(1,:)
        c12=c12+MATMUL(funny, rowf)*det*weights(i)/rho
        rowf(1,:)=derivf(2,:)
        c32=c32+MATMUL(funny, rowf)*det*weights(i)/rho
        rowf(1,:)=derivf(3,:)
        c42=c42+MATMUL(funny, rowf)*det*weights(i)/rho
        c21=c21+MATMUL(funnyf, row1)*det*weights(i)
    END DO
END DO

```

```

c23=c23+MATMUL(funnyf,row2)*det*weights(i)
c24=c24+MATMUL(funnyf,row3)*det*weights(i)
END DO gauss_points_1
CALL formupvw(storke_pp,iel,c11,c12,c21,c23,c32,c24,c42)
END DO elements_2
!----- build the preconditioner -----
diag_tmp=zero
elements_2a: DO iel=1,nels_pp; DO k=1,ntot
  diag_tmp(k,iel)=diag_tmp(k,iel)+storke_pp(k,k,iel); END DO
END DO elements_2a; CALL scatter(diag_pp,diag_tmp)
!----- prescribed values of velocity and pressure -----
DO i=1,fixed_freedoms_pp; k=no_pp(i)-ieq_start+1
  diag_pp(k)=diag_pp(k)+penalty
  b_pp(k)=diag_pp(k)*val(fixed_freedoms_start+i-1)
  store_pp(k)=diag_pp(k)
END DO
!----- solve the equations element-by-element using BiCGSTAB -----
!----- initialisation phase -----
IF(iters==1) x_pp=x0; pmul_pp=zero; y1_pp=zero; y_pp=x_pp
CALL gather(y_pp,pmul_pp)
elements_3: DO iel=1,nels_pp
  utemp_pp(:,iel)=MATMUL(storke_pp(:,:,iel),pmul_pp(:,iel))
END DO elements_3; CALL scatter(y1_pp,utemp_pp)
DO i=1,fixed_freedoms_pp; k=no_pp(i)-ieq_start+1
  y1_pp(k)=y_pp(k)*store_pp(k)
END DO; y_pp=y1_pp; rt_pp=b_pp-y_pp; r_pp=zero; r_pp(:,1)=rt_pp
u_pp=zero; gama=one; omega=one; k=0; norm_r=norm_p(rt_pp)
r0_norm=norm_r; error=one; cjitters=0
!----- BiCGSTAB(ell) iterations -----
bicg_iterations: DO; cjitters=cjitters+1
  cj_converged=error<cjtol; IF(cjitters==cjits.OR.cj_converged) EXIT
  gama=-omega*gama; y_pp=r_pp(:,1)
  DO j=1,ell
    rho1=DOT_PRODUCT_P(rt_pp,y_pp); beta=rho1/gama
    u_pp(:,1:j)=r_pp(:,1:j)-beta*u_pp(:,1:j)
    pmul_pp=zero; y_pp=u_pp(:,j); y1_pp=zero; CALL gather(y_pp,pmul_pp)
    elements_4: DO iel=1,nels_pp
      utemp_pp(:,iel)=MATMUL(storke_pp(:,:,iel),pmul_pp(:,iel))
    END DO elements_4; CALL scatter(y1_pp,utemp_pp)
    DO i=1,fixed_freedoms_pp; l=no_pp(i)-ieq_start+1
      y1_pp(l)=y_pp(l)*store_pp(l)
    END DO; y_pp=y1_pp; u_pp(:,j+1)=y_pp
    gama=DOT_PRODUCT_P(rt_pp,y_pp); alpha=rho1/gama
    x_pp=x_pp+alpha*u_pp(:,1)
    r_pp(:,1:j)=r_pp(:,1:j)-alpha*u_pp(:,2:j+1)
    pmul_pp=zero; y_pp=r_pp(:,j); y1_pp=zero; CALL gather(y_pp,pmul_pp)
    elements_5: DO iel=1,nels_pp
      utemp_pp(:,iel)=MATMUL(storke_pp(:,:,iel),pmul_pp(:,iel))
    END DO elements_5; CALL scatter(y1_pp,utemp_pp)
    DO i=1,fixed_freedoms_pp; l=no_pp(i)-ieq_start+1
      y1_pp(l)=y_pp(l)*store_pp(l)
    END DO; y_pp=y1_pp; r_pp(:,j+1)=y_pp
  END DO
  DO i=1,ell+1; DO j=1,ell+1
    GG(i,j)=DOT_PRODUCT_P(r_pp(:,i),r_pp(:,j))
  END DO; END DO
  CALL form_s(GG,ell,kappa,omega,Gamma,s)
  x_pp=x_pp-MATMUL(r_pp,s); r_pp(:,1)=MATMUL(r_pp,Gamma)

```

```

u_pp(:,1)=MATMUL(u_pp,Gamma); norm_r=norm_p(r_pp(:,1))
error=norm_r/r0_norm; k=k+1
END DO bicg_iterations
b_pp=x_pp-xold_pp; pp=norm_p(b_pp); cj_tot=cj_tot+cjitters
IF(numpe==it) THEN
    WRITE(11,'(A,E12.4)') "Norm of the error is:", pp
    WRITE(11,'(A,I6,A)') "It took BiCGSTAB(L) ", cjitters,
    " iterations to converge"; END IF
CALL checon_par(x_pp,tol,converged,xold_pp)
IF(converged.OR.iters==limit)EXIT
END DO iterations; timest(4)=elap_time()
DEALLOCATE(rt_pp,r_pp,u_pp,b_pp,diag_pp,xold_pp,y_pp,y1_pp,store_pp)
DEALLOCATE(storke_pp,pmul_pp)
!----- output results -----
IF(numpe==it) THEN
    WRITE(11,'(A)') "The pressure at the corner of the box is: "
    WRITE(11,'(A)') "Freedom Pressure "
    WRITE(11,'(I6,E12.4)') nres, x_pp(is)
    WRITE(11,'(A,I6)') "The total number of BiCGSTAB iterations was:",cj_tot
    WRITE(11,'(A,I5,A)') "The solution took",iters," iterations to converge"
    WRITE(11,'(A,F10.4)') "Time spent in solver was:",timest(4)-timest(3)
END IF
IF(numpe==1) THEN
    OPEN(12,file=argv(1:nlen)//".ensi.VEL",status='replace',
    action='write')
    WRITE(12,'(A)') "Alya Ensight Gold --- Vector per-node variable file"
    WRITE(12,'(A/A/A)') "part", " 1", "coordinates"
END IF
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
ALLOCATE(upvw_pp(nodes_pp*nodof),temp(nodes_pp)); upvw_pp=zero
temp=zero; utemp_pp=zero; CALL gather(x_pp(1:),utemp_pp)
CALL scatter_nodes_ns(npes,nn,nelss_pp,g_num_pp,nod,nodof,nodes_pp,
node_start,node_end,utemp_pp,upvw_pp,1)
DO i=1,nodof ; temp=zero
    IF(i/=2) THEN
        DO j=1,nodes_pp; k=i+(nodof*(j-1)); temp(j)=upvw_pp(k); END DO
        CALL dismsh_ensi_p(12,1,nodes_pp,npes,numpe,1,temp); END IF
END DO ; IF(numpe==1) CLOSE(12)
IF(numpe==it) THEN
    WRITE(11,'(A,F10.4)') "This analysis took :", elap_time()-timest(1)
    CLOSE(11); END IF; CALL SHUTDOWN()
END PROGRAM p126

```

The example used here is the cuboidal lid-driven cavity problem and as with the earlier programs in this chapter, the model is generated externally using p12meshgen (see Section 12.2.5). Loop elements\_2, with embedded gauss\_pts\_1, carries over from serial to parallel, the differences being due to the extension from 2D (serial) to 3D (parallel). For example, formupv (serial) becomes formupvw (parallel). After initialisation involving loop elements\_3, the BiCGStab iterations with loops elements\_4 and elements\_5 can be traced clearly between the two versions. In parallel, only the pressure at the corner of the box is printed in the summary results. Velocities at all the nodes are written out for visualisation using dismsh\_ensi\_p. Data are listed as Figure 12.33 with results as Figure 12.34.

The effect of problem size on the ‘average’ number of iterations to convergence is shown in Figure 12.35. Performance statistics are listed in Figure 12.36, together with

```

program
'p126'

iotype    nels nxe nze nip
'parafem' 8000 20 20 8

aa    bb    cc
0.05 0.05 0.05

visc rho tol limit
0.01 1.0 0.001 30

cjtol cjits penalty
1.0E-5 500 1.0E5

x0 ell kappa
1.0 4 0.0

```

**Figure 12.33** Data for Program 12.6 example

```

This job ran on      64 processes
There are 35721 nodes 28862 restrained and 96078 equations
Time to read input was:   1.3138
Time after setup was:   1.4518
Norm of the error is: 0.1529E+03
It took BiCGSTAB(L)   168 iterations to converge
Norm of the error is: 0.3548E+02
It took BiCGSTAB(L)   153 iterations to converge
Norm of the error is: 0.8622E+01
It took BiCGSTAB(L)   147 iterations to converge
Norm of the error is: 0.2214E+01
.
.
.
It took BiCGSTAB(L)   157 iterations to converge
Norm of the error is: 0.7124E-01
It took BiCGSTAB(L)   151 iterations to converge
The pressure at the corner of the box is:
Freedom Pressure
        481 0.1110E+01
The total number of BiCGSTAB iterations was: 1377
The solution took  9 iterations to converge
Time spent in solver was: 14.6808
This analysis took : 17.8113

```

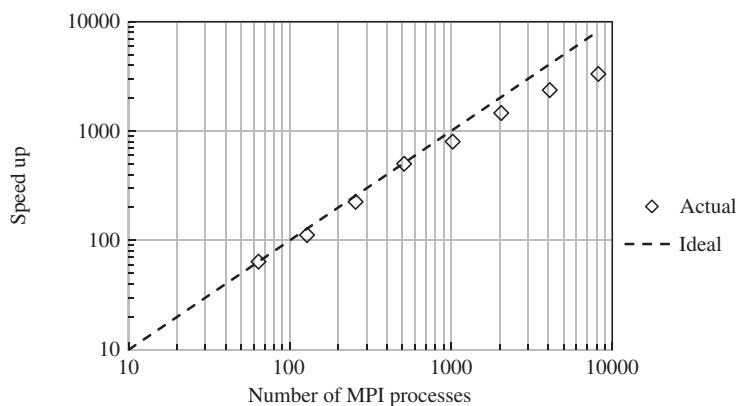
**Figure 12.34** Results from Program 12.6 example

iteration counts. Figure 12.36 shows that iteration count varies with number of processes used. This is due to the sensitivity of BiCGStab(l) to roundoff. In parallel, the summation order of distributed numerical operations (e.g., dot products) is different using different numbers of processes. This results in slightly different values for summations. Smith and Margetts (2006) show that BiCGStab(l) uses ‘extra’ iterations to overcome this round-off and always achieves the same ‘engineering’ answer. The result is *always* the same, regardless of the number of processes used, up to the value of tol defined by the user.

Elements	Iterations to convergence
512	672
1,000	713
1,728	900
8,000	1,385
64,000	2,508
125,000	3,313
1,000,000	5,897

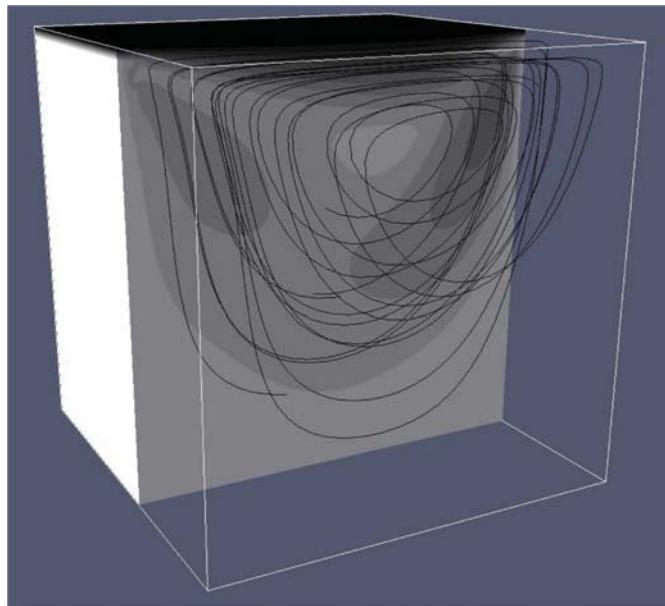
**Figure 12.35** Problem size vs. number of iterations: Program 12.6 (Cray XE6)

Mesh	No of MPI Processes	Analysis Time (secs)	Iterations
20x20x20	8	88	1307
	16	47	1314
	32	28	1396
	32	18	1353

**Figure 12.36** Performance statistics: Program 12.6 (SGI Intel Cluster)**Figure 12.37** Speed-up vs. number of processes with 12.7 million equations: Program 12.6 (Cray XE6)

Speedup vs. number of processes for a larger data set is shown in Figure 12.37. A typical display is shown in Figure 12.38.

Program 12.6 can be easily modified to solve the equations of magnetohydrodynamics (Margetts, 2002) and has been adapted to study the generation of the Earth's magnetic field by Chan *et al.* (2006, 2007).



**Figure 12.38** Contours of velocity magnitude and streamlines using ParaView: Program 12.6

### Program 12.7 Three-dimensional analysis of Biot poro elastic solid. Incremental version. Compare Program 9.5

```

PROGRAM p127
!-----
!      Program 12.7 3-D consolidation of a cuboidal Biot elastic
!      solid using 20-node solid hexahedral elements coupled to 8-node
!      fluid elements-incremental version-parallel pcg version - biot_cube
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE loading
USE timing; USE maths; USE gather_scatter; USE new_library; USE geometry
USE input; IMPLICIT NONE
! neq,ntot are now global variables - not declared
INTEGER::nxr,nyr,nzr,nn,rr,nip,nodof=4,nod=20,nodf=8,nst=6,ndim=3,i,j,k,&
  1iel,ns,nstep,cjitters,cjits,loaded_freedoms,loaded_freedoms_pp,n_t,&
  neq_temp,nn_temp,nle,nlen,nels,partitioner=1,nodof,ielpe,npes_pp,&
  loaded_freedoms_start,nlfp,nls,is,it,nres
REAL(iwp)::kx,ky,kz,e,v,det,dtim,theta,real_time,up,alpha,beta,cjtol,aa,&
  bb,cc,q
REAL(iwp),PARAMETER::zero=0._iwp
LOGICAL::cj_converged; CHARACTER(LEN=15)::element='hexahedron'
CHARACTER(LEN=50)::argv
!----- dynamic arrays-----
REAL(iwp),ALLOCATABLE::dee(:,:,:),points(:,:,:),coord(:,:,:),derivf(:,:),
  jac(:,:,:),kay(:,:,:),der(:,:,:),deriv(:,:,:),weights(:),derf(:,:,:),funf(:),
  & coordf(:,:,:),bee(:,:,:),km(:,:,:),eld(:),sigma(:),kc(:,:,:),ke(:,:),
  & g_coord_pp(:,:,:,:),kd(:,:,:),fun(:),c(:,:,:),loads_pp(:),pmul_pp(:,:),
  & storke_pp(:,:,:,:),ans_pp(:),volf(:,:,:),p_pp(:),x_pp(:),xnew_pp(:),
  & u_pp(:),eld_pp(:,:,:),diag_precon_pp(:),diag_precon_tmp(:,:,:),d_pp(:),
  & utemp_pp(:,:,:),storkc_pp(:,:,:,:),val(:),vol(:),timestep(:),al(:),lf(:,:),
  & INTEGER,ALLOCATABLE::rest(:,:,:),g(:),num(:),g_g_pp(:,:,:),g_num_pp(:,:),
  &
  
```

```

g_t(:,),no(:,),no_pp_temp(:,),no_pp(:)
!----- input and initialisation-----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p127(argv,numpe,nels,nxe,nze,aa,bb,cc,nip,kx,ky,kz,e,v,dtim,
  nstep,theta,cjits,cjtol,nlfp)
CALL calc_nels_pp(argv,nels,npes,numpe,partitioner,nels_pp)
ndof=nod*ndim; ntot=ndof+nodf; n_t=nod*nodf
nye=nels/nxe/nze; nle=nxe/5; loaded_freedoms=3*nle*nle+4*nle+1
nr=3*nxe*nye*nze+4*(nxe*nye+nye*nze+nze*nxe)+nxe+nye+nze+2
nres=2*nxe*2+nxe*2+1
ALLOCATE(points(nip,ndim),coord(nod,ndim),derivf(ndim,nodf),
  jac(ndim,ndim),kay(ndim,ndim),der(ndim,nod),deriv(ndim,nod),
  derf(ndim,nodf),funf(nodf),coordf(nodf,ndim),bee(nst,n dof),
  km(ndof,ndof),eld(ndof),sigma(nst),kc(nodf,nodf),weights(nip),
  g_g_pp(ntot,nels_pp),diag_precon_tmp(ntot,nels_pp),ke(ntot,ntot),
  kd(ntot,ntot),fun(nod),c(ndof,nodf),g_t(n_t),vol(ndof),
  rest(nr,nodof+1),g(ntot),volf(ndof,nodf),g_coord_pp(nod,ndim,nels_pp),
  g_num_pp(nod,nels_pp),num(nod),storke_pp(ntot,ntot,nels_pp),
  storkc_pp(nodf,nodf,nels_pp),pmul_pp(ntot,nels_pp),dee(nst,nst),
  utemp_pp(ntot,nels_pp),eld_pp(ntot,nels_pp),no(loaded_freedoms),
  val(loaded_freedoms),no_pp_temp(loaded_freedoms))
kay=0.0_iwp; kay(1,1)=kx; kay(2,2)=ky; kay(3,3)=kz
CALL biot_cube_bc20(nxe,nye,nze,rest); CALL rearrange(rest)
CALL cell_loading(nxe,nze,nle,no, val); val=-val*aa*bb/12._iwp
CALL sample(element,points,weights); CALL deemat(dee,e,v)
ALLOCATE(lf(2,nlfp)); lf=zero; CALL read_lf(argv,numpe,lf)
nls=FLOOR(lf(1,nlfp)/dtim); IF(nstep>nls)nstep=nls
ALLOCATE(al(nstep)); al=zero; CALL load_function(lf,dtim,al)
!----- loop the elements to set up global arrays -----
neq_temp=0; nn_temp=0; ielpe=iel_start
elements_1: DO iel=1,nels_pp
  CALL geometry_20bxz(ielpe,nxe,nze,aa,bb,cc,coord,num)
  CALL find_g3(num,g_t,rest); CALL g_t_g(nod,g_t,g)
  g_coord_pp(:,:,iel)=coord; g_g_pp(:,:,iel)=g; ielpe=ielpe+1
  i=MAXVAL(g); j=MAXVAL(num); g_num_pp(:,:,iel)=num
  IF(i>neq_temp)neq_temp=i; IF(j>nn_temp)nn_temp=j
END DO elements_1
neq=max_p(neq_temp); nn=max_p(nn_temp); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
nres=6*nxe+1
DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN; it=numpe; is=i; END IF; END DO
IF(numpe==it)THEN
  OPEN(11,FILE=argv(1:nlen)//'.res',STATUS='REPLACE',ACTION='WRITE')
  WRITE(11,'(A,I5,A)')"This job ran on ",npes," processes"
  WRITE(11,'(A,3(I8,A))')"There are ",nn," nodes",nr," restrained and ",&
    neq," equations"
  WRITE(11,'(A,F10.4)')"Time after setup is:",elap_time()-timest(1)
END IF
ALLOCATE(loads_pp(neq_pp),ans_pp(neq_pp),p_pp(neq_pp),x_pp(neq_pp),
  xnew_pp(neq_pp),u_pp(neq_pp),diag_precon_pp(neq_pp),d_pp(neq_pp))
loads_pp=.0_iwp; p_pp=.0_iwp; xnew_pp=.0_iwp; diag_precon_pp=.0_iwp
diag_precon_tmp=.0_iwp
!----- element stiffness integration, storage and preconditioner -----
elements_2: DO iel=1,nels_pp
  coord=g_coord_pp(:,:,iel); coordf(1:4,:)=coord(1:7:2,:)
  coordf(5:8,:)=coord(13:20:2,:); km=zero; c=zero; kc=zero
  gauss_points_1: DO i=1,nip
    CALL shape_der(der,points,i); jac=MATMUL(der,coord)
    det=determinant(jac); CALL invert(jac)

```

```

deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
vol(:)=bee(1,:)+bee(2,:)+bee(3,:)
km=km+MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)
!----- now the fluid contribution -----
CALL shape_fun(funf,points,i); CALL shape_der(derf,points,i)
derivf=MATMUL(jac,derf)
kc=kc+
  MATMUL(MATMUL(TRANSPOSE(derivf),kay),derivf)*det*weights(i)*dtim
DO l=1,nodf; volf(:,l)=vol(:)*funf(l); END DO
c=c+volf*det*weights(i)
END DO gauss_points_1
storkc_pp(:,:,iel)=kc; CALL formke(km,kc,c,ke,theta)
storke_pp(:,:,iel)=ke
DO k=1,ndof
  diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel)+theta*km(k,k)
END DO
DO k=1,ndof
  diag_precon_tmp(ndof+k,iel)=diag_precon_tmp(ndof+k,iel) &
  -theta*theta*kc(k,k)
END DO
END DO elements_2
CALL scatter(diag_precon_pp,diag_precon_tmp)
diag_precon_pp=1._iwp/diag_precon_pp
DEALLOCATE(diag_precon_tmp)
!-----loaded freedoms -----
no_pp_temp=0
CALL reindex(ieq_start,no,no_pp_temp,loaded_freedoms_pp, &
  loaded_freedoms_start,neq_pp)
ALLOCATE(no_pp(1:loaded_freedoms_pp)); no_pp=0
no_pp=no_pp_temp(1:loaded_freedoms_pp); DEALLOCATE(no_pp_temp)
!----- enter the time-stepping loop -----
real_time=zero
time_steps: DO ns=1,nstep
  real_time=real_time+dtim
  IF(numpe==1) WRITE(11,'(A,E12.4)') "The time is", real_time
  pmul_pp=zero; utemp_pp=zero; ans_pp=zero
  CALL gather.loads_pp,pmul_pp)
elements_3: DO iel=1,nels_pp
  utemp_pp(ndof+1:,iel)=MATMUL(storkc_pp(:,:,iel),pmul_pp(ndof+1:,iel))
END DO elements_3; CALL scatter(ans_pp,utemp_pp)
!----- ramp loading -----
IF.loaded_freedoms_pp>0) THEN
  DO i=1,loaded_freedoms_pp; j=no_pp(i)-ieq_start+1
    ans_pp(j)=val.loaded_freedoms_start+i-1)*al(ns)
  END DO
END IF
d_pp=diag_precon_pp*ans_pp; p_pp=d_pp
x_pp=zero ! depends on starting x = zero
!----- solve the simultaneous equations by pcg -----
cjiters=0
conjugate_gradients: DO
  cjiters=cjiters+1; u_pp=zero; pmul_pp=zero; utemp_pp=zero
  CALL gather(p_pp,pmul_pp)
elements_4: DO iel=1,nels_pp
  utemp_pp(:,iel)=MATMUL(storke_pp(:,:,iel),pmul_pp(:,iel))
END DO elements_4; CALL scatter(u_pp,utemp_pp)
!----- pcg process -----
up=DOT_PRODUCT_P(ans_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)

```

```

xnew_pp=x_pp+p_pp*alpha; ans_pp=ans_pp-u_pp*alpha
d_pp=diag_precon_pp*ans_pp; beta=DOT_PRODUCT_P(ans_pp,d_pp)/up
p_pp=d_pp+p_pp*beta
CALL checon_par(xnew_pp,cjtol,cj_converged,x_pp)
IF(cj_converged.OR.cjiter==cjits)EXIT
END DO conjugate_gradients
ans_pp=xnew_pp; loads_pp=loads_pp+ans_pp
IF(numpe==it)THEN
  WRITE(11,'(A,I5,A)')
    "Conjugate gradients took ",cjiter," iterations to converge"
  WRITE(11,'(A)')" The vertical displacement and pore pressure is      :"
  WRITE(11,'(2E12.4)')loads_pp(is), loads_pp(is+1)
END IF
!----- recover stresses at gauss-points -----
eld_pp=zero; CALL gather(loads_pp,eld_pp)
iel=1; coord=g_coord_pp(:,:,iel); eld=eld_pp(:ndof,iel)
IF(numpe==it)WRITE(11,'(A,I5,A)') &
  "The Gauss Point effective stresses for element",iel," are"
gauss_pts_2: DO i=1,nip
  CALL shape_der(der,points,i); jac=MATMUL(der,coord)
  CALL invert(jac); deriv=MATMUL(jac,der)
  CALL beemat(bee,deriv); sigma=MATMUL(dee,MATMUL(bee,eld))
  IF(numpe==it.AND.i==1)THEN
    WRITE(11,'(A,I5)')"Point ",i
    WRITE(11,'(6E12.4)')sigma
  END IF
END DO gauss_pts_2
END DO time_steps
IF(numpe==it) THEN; WRITE(11,'(A,F10.4)')"This analysis took:", &
  elap_time()-timest(1); CLOSE(11); END IF; CALL SHUTDOWN()
END PROGRAM p127

```

In contrast to the other programs in this chapter, Program 12.7 shows how to generate the geometry of the model, in parallel, inside the program itself. Again the main differences between serial and parallel programs relate to the change in geometry from 2D to 3D. Loop `elements_1` uses `geometry_20bxz` in place of `geom_rect` in serial and loop `elements_2` with embedded `gauss_pts_1` is almost identical although the parallel version assumes constant element properties. Ramp loading is also assumed rather than the general pattern allowed in serial (2D). Loop `elements_4` carries over from serial to parallel but in the latter case only a few surface displacements are printed and only the stresses in the ‘first’ (central surface) element are computed and printed. Data are listed as Figure 12.39 with results as Figure 12.40 and performance statistics as Figure 12.41.

nels	nxe	nze		
64000	40	40		
aa	bb	cc	nip	
0.25	0.25	0.25	8	
kx	ky	kz	e	v
1.0	1.0	1.0	1.0	0.0
dtim	nstep	theta		
1.0	20	1.0		
cjits	cjtol			
1000	.00001			

**Figure 12.39** Data for Program 12.7 example

```

This job ran on      32 processes
There are    270641 nodes  211322 restrained and    844760  equations
Time after setup is:  0.2350
The time is  0.1000E+01
Conjugate gradients took   662 iterations to converge
  The nodal displacements and porepressures are   :
  -0.2578E+00 -0.2896E-02 -0.2576E+00 -0.5815E-02
The Gauss Point effective stresses for element    1 are
Point    1
  -0.2168E-01 -0.2168E-01 -0.9630E-01  0.2128E-03 -0.1041E-03 -0.1041E-03
The time is  0.2000E+01
Conjugate gradients took   605 iterations to converge
  The nodal displacements and porepressures are   :
  -0.5474E+00 -0.6964E-02 -0.5468E+00 -0.1395E-01
The Gauss Point effective stresses for element    1 are
Point    1
  -0.5223E-01 -0.5223E-01 -0.1947E+00  0.4134E-03  0.1596E-03  0.1596E-03
.
.
.
The time is  0.2000E+02
Conjugate gradients took   702 iterations to converge
  The nodal displacements and porepressures are   :
  -0.4075E+01 -0.7079E-01 -0.4071E+01 -0.1416E+00
The Gauss Point effective stresses for element    1 are
Point    1
  -0.5391E+00 -0.5391E+00 -0.9936E+00  0.3764E-02  0.1407E-03  0.1407E-03
This analysis took:  200.9534

```

**Figure 12.40** Results from Program 12.7 example

Mesh	No of MPI Processes	Analysis Time (secs)
40x40x40	16	397
	32	200
	64	138
	128	94

**Figure 12.41** Performance statistics: Program 12.7 (SGI Intel Cluster)

## Program 12.8 Eigenvalue analysis of three-dimensional elastic solid. Compare Program 10.3

```

PROGRAM p128
!-----
! Program 12.8 Eigenvalues and eigenvectors of a cuboidal elastic
!           solid in 3d using uniform 8-node hexahedral elements
!           for lumped mass this is done element by element : parallel
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE new_library; USE eigen; IMPLICIT NONE
! neq,ntot are now global variables - not declared
INTEGER,PARAMETER::nodedof=3,nod=8,nst=6,ndim=3
INTEGER::nels,nn,nr,nip,i,j,k,iel,nmodes,jflag,lalfa,leig,lx,lz,
         &

```

```

iters,nlen,ndof,partitioner,meshgen,npes_pp,neig=0,iflag=-1,lp=11,      &
nodes_end,nodes_start,nodes_pp
REAL(iwp)::rho,e,v,det,el,er,acc ; REAL(iwp),PARAMETER::zero=0.0_iwp
CHARACTER(LEN=15)::element='hexahedron';  CHARACTER(LEN=50)::argv
CHARACTER(LEN=6)::ch
!----- dynamic arrays-----
REAL(iwp),ALLOCATABLE::points(:,:),dee(:,:),vdiag_pp(:,eig(:,del(:,&
fun(:,jac(:,der(:,deriv(:,weights(:,bee(:,val_pp(:,&
emmm(:,ecm(:,utemp_pp(:,ua_pp(:,storkm_pp(:,timest(:,&
udiag_pp(:,diag_pp(:,alfa(:,beta(:,w1_pp(:,y_pp(:,z_pp(:,&
pmul_pp(:,v_store_pp(:,g_coord_pp(:, :,diag_tmp(:,x(:,&
va_pp(:,eigv_pp(:,temp(:,&
INTEGER,ALLOCATABLE::rest(:,g_num_pp(:,g_g_pp (:,:,nu(:,jeig(:,&
!-----input and initialisation-----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p128(argv,numpe,acc,e,el,er,lalfa,leig,lx,lz,meshgen,nels,&
nip,nmodes,nn,nr,partitioner,rho,v)
CALL calc_nels_pp(argv,nels,npes,numpe,partitioner,nels_pp)
ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod, nels_pp),g_coord_pp(nod,ndim, nels_pp),&
rest(nr,nodof+1)); g_num_pp=0; g_coord_pp=zero; rest=0
CALL read_g_num_pp(argv,iel_start,nn,npes,numpe,g_num_pp)
IF(meshgen == 2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
CALL read_rest(argv,numpe,rest); timest(2)=elap_time()
ALLOCATE(points(nip,ndim),pmul_pp(ntot,nelss_pp),fun(nod),dee(nst,nst),&
jac(ndim,ndim),weights(nip),der(ndim,nod),deriv(ndim,nod),x(lx),&
g_g_pp(ntot,nelss_pp),ecm(ntot,ntot),eig(leig),del(lx),nu(lx),&
jeig(2,leig),alfa(lalfa),beta(lalfa),z_pp(lz,leig),bee(nst,ntot),&
utemp_pp(ntot,nelss_pp),emm(ntot,ntot),diag_tmp(ntot,nelss_pp),&
storkm_pp(ntot,ntot,nelss_pp))
!----- find the steering array and equations per process -----
CALL rearrange(rest); g_g_pp=0; neq=0
elements_0: DO iel=1,nelss_pp
    CALL find_g3(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_0
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npess_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
IF(numpe==1) THEN
    OPEN(11,FILE=argv(1:nlen)//".res",STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I7,A)') "This job ran on ",npes," processes"
    WRITE(11,'(A,3(I12,A))') "There are ",nn," nodes", nr, &
        " restrained and ",neq," equations"
    WRITE(11,'(A,F10.4)') "Time to read input is:",timest(2)-timest(1)
    WRITE(11,'(A,F10.4)') "Time after setup is:",elap_time()-timest(1)
END IF
ALLOCATE(ua_pp(neq_pp),va_pp(neq_pp),vdiag_pp(neq_pp),udiag_pp(neq_pp),&
v_store_pp(neq_pp,lalfa),diag_pp(neq_pp),w1_pp(neq_pp),&
y_pp(neq_pp,leig)); ua_pp=zero; va_pp=zero; eig=zero
diag_tmp=zero; jeig=0; x=zero; del=zero; nu=0; alfa=zero; beta=zero
diag_pp=zero; udiag_pp=zero; w1_pp=zero; y_pp=zero; z_pp=zero
CALL sample(element,points,weights); CALL deemat(dee,e,v)
!----- element stiffness integration and assembly-----
elements_2: DO iel=1,nelss_pp; emm=zero
    integrating_pts_1: DO i=1,nip
        CALL shape_fun(fun,points,i); CALL shape_der(der,points,i)
        jac=MATMUL(der,g_coord_pp(:, :,iel)); det=determinant(jac)

```

```

CALL invert(jac); deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
storkm_pp(:,:,iel)=storkm_pp(:,:,iel)+  

    MATMUL(MATMUL(TRANSPOSE(bee),dee),bee)*det*weights(i)  

    CALL ecmat(ecm,fun,ntot,nodof); emm=emm+ecm*det*weights(i)*rho  

END DO integrating_pts_1  

DO k=1,ntot; diag_tmp(k,iel)=diag_tmp(k,iel)+sum(emm(k,:)); END DO  

END DO elements_2  

CALL scatter(diag_pp,diag_tmp); DEALLOCATE(diag_tmp)  

!----- find eigenvalues -----  

diag_pp=1._iwp/sqrt(diag_pp) ! diag_pp holds l**(-1/2)  

DO iters=1,lalpha  

    CALL lancz1(neq_pp,el,er,acc,leig,lp,iflag,ua_pp,va_pp,eig, &  

        jeig,neig,x,del,nu,alfa,beta,v_store_pp)  

    IF(iflag==0) EXIT  

    IF(iflag>1)THEN;  

        IF(numpe==1)THEN  

            WRITE(11,'(A,I5)')  

            " Lancz1 is signalling failure, with iflag = ",iflag; EXIT  

        END IF  

    END IF  

!---- iflag = 1 therefore form u + a * v (done element by element) ----  

vdiag_pp=va_pp; vdiag_pp=vdiag_pp*diag_pp ! vdiag is l**(-1/2).va  

udiag_pp=zero; pmul_pp=zero  

CALL gather(vdiag_pp,pmul_pp)  

elements_3: DO iel=1,nels_pp  

    utemp_pp(:,iel) = MATMUL(storkm_pp(:,:,iel),pmul_pp(:,iel))  

END DO elements_3  

CALL scatter(udiag_pp,utemp_pp) ! udiag is A.l**(-1/2).va  

udiag_pp=udiag_pp*diag_pp; ua_pp=ua_pp+udiag_pp  

END DO  

!----- iflag = 0 therefore write out the spectrum -----  

IF(numpe==1)THEN  

    WRITE(11,'(2(A,E12.4))')"The range is",el," to ",er  

    WRITE(11,'(A,I8,A)')"There are ",neig," eigenvalues in the range"  

    WRITE(11,'(A,I8,A)')"It took ",iters," iterations"  

    WRITE(11,'(A)')"The eigenvalues are:"; WRITE(11,'(6E12.4)')eig(1:neig)  

END IF  

! calculate the eigenvectors  

IF(neig>10)neig=10  

CALL lancz2(neq_pp,lalpha,lp,eig,jeig,neig,alfa,beta,lz,jflag,y_pp, &  

    w1_pp,z_pp,v_store_pp)  

!----- if jflag is zero calculate the eigenvectors -----  

IF(jflag==0)THEN; timest(3)=elap_time()  

    CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)  

    ALLOCATE(eigv_pp(nodes_pp*ndim),temp(nodes_pp))  

    eigv_pp=zero; utemp_pp=zero; temp=0  

    IF(numpe==1) WRITE(11,'(A)')"The eigenvectors are :"  

    DO i=1,nmodes  

        udiag_pp(:)=y_pp(:,i); udiag_pp=udiag_pp*diag_pp  

        IF(numpe==1) THEN  

            WRITE(11,'("Eigenvector number ",I4," is: ")')i  

            WRITE(11,'(6E12.4)')udiag_pp(1:6)  

            WRITE(ch,'(I6.6)') i  

            OPEN(12,file=argv(1:nlen)//".ensi.EIGV-"/ch,status='replace', &  

                action='write'); WRITE(12,'(A)')  

                "Alya Ensight Gold --- Vector per-node variable file"  

            WRITE(12,'(A/A/A)') "part", " 1", "coordinates"  

        END IF

```

```

CALL gather(udiag_pp(1:),utemp_pp); eigv_pp=zero
CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,ndim,nodes_pp,      &
                  node_start,node_end,utemp_pp,eigv_pp,1)
DO j=1,ndim ; temp=zero
  DO l=1,nodes_pp; k=j+(ndim*(l-1)); temp(l)=eigv_pp(k); END DO
  CALL dismsh_ensi_p(12,l,nodes_pp,npes,numpe,l,temp)
END DO ; IF(numpe==1) CLOSE(12)
END DO
IF(numpe==1) THEN
  WRITE(11,'(A,F10.4)')"Writing out eigenvectors took:",          &
    elap_time()-timest(3)
END IF
ELSE
  IF(numpe==1) THEN ! lancz2 fails
    WRITE(11,'(A,I5)')"Lancz2 is signalling failure with jflag= ", jflag
  END IF
END IF
IF(numpe==1)WRITE(11,'(A,F10.4)')
  "This analysis took:",elap_time()-timest(1)
CALL SHUTDOWN()
END PROGRAM p128

```

Loop elements\_2 with embedded loop integrating\_points\_1 clearly carries over from serial to parallel although the parallel version assumes uniform element properties throughout the mesh. In all other aspects the serial and parallel programs are essentially identical, although of course the serial and parallel solution algorithm libraries are not interchangeable. Data are listed as Figure 12.42 involving just over a million equations with results as Figure 12.43 and some performance statistics as Figure 12.44. A plot of the eigenvectors using ParaView is shown in Figure 12.45. The greyscale shading indicates the magnitude of displacement. For a larger data set, Figure 12.46 shows performance measured on a Cray XE6 computer. Note the time taken for the ASCII read and write is significant compared with the total analysis time, and this program would

```

program
'p128'

itype    nels    nxe nze nip
'parafem' 320000 40 40 8

aa      bb      cc
0.025 0.025 0.025

rho e   v
1.0 1.0 0.3

nmodes
5

el er
0.0 1.0

lalfa leig lx  lz   acc
5000  20    100 5000 1.0E-9

```

**Figure 12.42** Data for Program 12.8 example

```

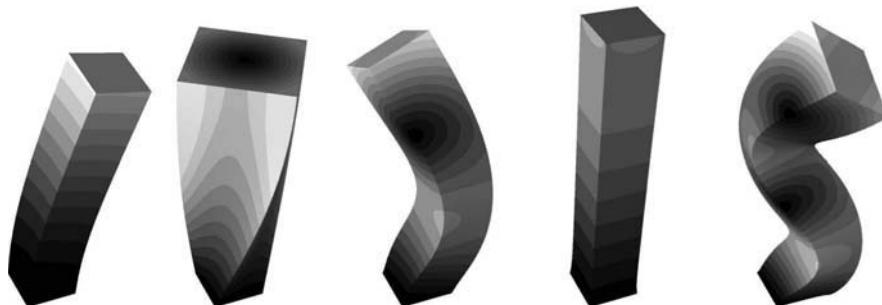
This job ran on      128 processes
There are  337881 nodes  1681 restrained and 1008600 equations
Time to read input is:   1.8921
Time after setup is:   1.9921
The range is  0.0000E+00  to  0.1000E+01
There are      9 eigenvalues in the range
It took     1608 iterations
The eigenvalues are:
  0.1579E-02  0.3220E-01  0.4588E-01  0.9975E-01  0.2608E+00  0.2897E+00
  0.7213E+00  0.8042E+00  0.8872E+00
The eigenvectors are :
Eigenvector number    1 is:
 -0.2972E-03 -0.3987E-06 -0.2967E-03 -0.3242E-03  0.1163E-03 -0.2509E-03
Eigenvector number    2 is:
  0.2099E-02  0.2217E-09  0.2099E-02  0.2388E-02 -0.2160E-02  0.3068E-02
Eigenvector number    3 is:
 -0.7210E-03  0.4090E-06 -0.7215E-03 -0.7472E-03 -0.2578E-03 -0.6571E-04
Eigenvector number    4 is:
  0.7434E-02  0.1133E-01 -0.7434E-02  0.4811E-02  0.8637E-02 -0.7253E-02
Eigenvector number    5 is:
 -0.2356E-04  0.2581E-04 -0.5712E-04  0.1221E-03 -0.2084E-02  0.2082E-02
Writing out eigenvectors took:   4.6243
This analysis took:   15.7490

```

**Figure 12.43** Results from Program 12.8 example

Mesh	No of MPI Processes	Analysis Time (secs)
40x200x40	16	81
	32	42
	64	25
	128	16

**Figure 12.44** Performance statistics: Program 12.8 (Cray XE6)



**Figure 12.45** ParaView plot of eigenvectors for Program 12.8 example

8.6M elements, 26M equations (time in seconds)			
# Processes	128	256	512
Read/Setup	54	47	43
Find Eigenvalues	424	212	107
Write Eigenvectors	93	95	96
Total	571	354	246
Iterations	4840	4810	4782

**Figure 12.46** Larger data set: Program 12.8 (Cray XE6)

benefit from the use of files in BINARY format (see Program 12.4). This data set required a large amount of memory per process, so the runs were carried out using only two cores on each 16-core processor (see Section 12.1).

## Program 12.9 Forced vibration analysis of a three-dimensional elastic solid. Implicit integration in time. Compare Program 11.7

```

PROGRAM p129
!-----
! Program 12.9 Forced vibration of a 3d elastic solid
!           Lumped or consistent mass
!           Implicit integration by theta method : parallel version
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE new_library; IMPLICIT NONE
! neq,ntot are now global variables - not declared
INTEGER,PARAMETER::nodof=3,nst=6
INTEGER::nn,nr,nip,nod,i,j,k,l,iel,ndim=3,nstep,npri,iters,limit,ndof, &
  nels,npes_pp,node_end,node_start,nodes_pp,loaded_nodes,nlen,nres, &
  meshgen,partitioner,it,is
REAL(iwp),PARAMETER::zero=0.0_iwp
REAL(iwp)::e,v,det,rho,alpha1,beta1,omega,theta,period,pi,dtim,volume, &
  c1,c2,c3,c4,real_time,tol,big,up,alpha,beta,tload
CHARACTER(LEN=15)::element; CHARACTER(LEN=50)::argv; CHARACTER(LEN=6)::ch
LOGICAL::consistent=.TRUE.,converged
!----- dynamic arrays -----
REAL(iwp),ALLOCATABLE::loads_pp(:),points(:,:),dee(:,:),fun(:),jac(:,:),&
  der(:,:),deriv(:,:),weights(:,bee(:,:)),g_coord_pp(:,:,:),fext_pp(:), &
  x1_pp(:),d1x1_pp(:),d2x1_pp(:),emm(:,:),ecm(:,:),x0_pp(:),d1x0_pp(:), &
  d2x0_pp(:),store_km_pp(:,:,:),vu_pp(:),store_mm_pp(:,:,:),u_pp(:), &
  p_pp(:),d_pp(:),x_pp(:),xnew_pp(:),pmul_pp(:,:),utemp_pp(:,:),temp(:), &
  diag_precon_pp(:),diag_precon_tmp(:,:),temp_pp(:,:,:),disp_pp(:), &
  val(:,:),timest(:,eld_pp(:,:))
INTEGER,ALLOCATABLE::rest(:,:),g_num_pp(:,:),g_g_pp(:,:),node(:)
!----- input and initialisation -----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p129(argv,numpe,alpha1,beta1,e,element,limit,loaded_nodes, &
  meshgen,nels,nip,nn,nod,npri,lr,nres,nstep,omega,partitioner,rho, &
  theta,tol,v)
CALL calc_nels_pp(argv,nel,nlp,npes,numpe,partitioner,nel_pp)
ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod, nels_pp),g_coord_pp(nod,ndim,nel_pp), &
  rest(nr,nodof+1)); g_num_pp=0; g_coord_pp=zero; rest=0
CALL read_g_num_pp(argv,iel_start,nel,nn,numpe,g_num_pp)
IF(meshgen==2) CALL abaqus2sg(element,g_num_pp)
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
CALL read_rest(argv,numpe,rest)
ALLOCATE(points(nip,ndim),bee(nst,ntot),dee(nst,nst),jac(ndim,ndim), &
  weights(nip),der(ndim,nod),deriv(ndim,nod),g_g_pp(ntot,nel_pp), &
  store_km_pp(ntot,ntot,nel_pp),utemp_pp(ntot,nel_pp),ecm(ntot,ntot), &
  pmul_pp(ntot,nel_pp),store_mm_pp(ntot,ntot,nel_pp),emm(ntot,ntot), &
  temp_pp(ntot,ntot,nel_pp),diag_precon_tmp(ntot,nel_pp),fun(ntot))
!----- find the steering array and equations per process -----
CALL rearrange(rest); g_g_pp=0; neq=0
elements_1: DO iel = 1, nels_pp

```

```

CALL find_g3(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_1
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN;it=numpe;is=i;END IF;END DO
IF(numpe==it) THEN
  OPEN(11,FILE=argv(1:nlen)//".res",STATUS='REPLACE',ACTION='WRITE')
  WRITE(11,'(A,I6,A)') "This job ran on ",npes," processes"
  WRITE(11,'(A,3(I12,A))') "There are ",nn," nodes ",nr,
  & " restrained and ", neq, " equations"
  WRITE(11,'(A,F10.4)') "Time after setup was:", elap_time()-timest(1)
END IF
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
ALLOCATE(disp_pp(nodes_pp*ndim),temp(nodes_pp),eld_pp(ntot,nels_pp))
disp_pp=zero; eld_pp=zero; temp=0
ALLOCATE(x0_pp(neq_pp),d1x0_pp(neq_pp),x1_pp(neq_pp),vu_pp(neq_pp),
  & diag_precon_pp(neq_pp),u_pp(neq_pp),d2x0_pp(neq_pp),loads_pp(neq_pp),
  & d1x1_pp(neq_pp),d2x1_pp(neq_pp),d_pp(neq_pp),p_pp(neq_pp),
  & x_pp(neq_pp),xnew_pp(neq_pp),fext_pp(neq_pp))
x0_pp=zero; d1x0_pp=zero; x1_pp=zero; vu_pp=zero; diag_precon_pp=zero
u_pp=zero; d2x0_pp=zero; loads_pp=zero; d1x1_pp=zero; d2x1_pp=zero
d_pp=zero; p_pp=zero; x_pp=zero; xnew_pp=zero; fext_pp=zero
!--- element stiffness and mass integration, storage and preconditioner --
CALL deemat(dee,e,v); CALL sample(element,points,weights)
store_km_pp=zero; store_mm_pp=zero; diag_precon_tmp=zero
pi=ACOS(-1._iwp); period=2._iwp*pi/omega; dtim=period/20._iwp
c1=(1._iwp-theta)*dtim; c2=beta1-c1; c3=alpha1+1._iwp/(theta * dtim)
c4=beta1+theta*dtim
elements_2: DO iel=1,nels_pp;    volume=zero; emm=zero; ecm=zero
  gauss_points_1: DO i=1,nip
    CALL shape_der(der,points,i); jac=MATMUL(der,g_coord_pp(:,:,iel))
    det=determinant(jac); CALL invert(jac); deriv=matmul(jac,der)
    CALL beemat(bee,deriv)
    store_km_pp(:,:,iel)=store_km_pp(:,:,iel) +
      MATMUL(MATMUL(TRANSPOSE(bee),dee),bee) *
      & det*weights(i)
    volume=volume+det*weights(i); CALL shape_fun(fun,points,i)
    IF(consistent)THEN; CALL ecmat(ecm,fun,ntot,nodof)
      ecm=ecm*det*weights(i)*rho; emm=emm+ecm
    END IF
  END DO gauss_points_1
  IF(.NOT.consistent)THEN
    DO i=1,ntot; emm(i,i)=volume*rho/13._iwp; END DO
    DO i=1,19,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
    DO i=2,20,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
    DO i=3,21,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
    DO i=37,55,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
    DO i=38,56,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
    DO i=39,57,6; emm(i,i)=emm(4,4)*.125_iwp; END DO
  END IF
  store_mm_pp(:,:,iel)=emm
END DO elements_2
diag_precon_tmp=zero
elements_2a: DO iel=1,nels_pp
  DO k=1,ntot
    diag_precon_tmp(k,iel)=diag_precon_tmp(k,iel) +
      & store_mm_pp(k,k,iel)*c3+store_km_pp(k,k,iel)*c4
  END DO
END DO elements_2a; CALL scatter(diag_precon_pp,diag_precon_tmp)

```

```

diag_precon_pp=1._iwp/diag_precon_pp; DEALLOCATE(diag_precon_tmp)
!----- loads -----
IF.loaded_nodes>0) THEN
  ALLOCATE(node.loaded_nodes),val(ndim,loaded_nodes); val=zero; node=0
  CALL read_loads(argv,numpe,node,val)
  CALL load(g_g_pp,g_num_pp,node,val,fext_pp(1:))
  tload=SUM_P(fext_pp(1:)); DEALLOCATE(node,val)
END IF
!----- initial conditions -----
x0_pp=zero; d1x0_pp=zero; d2x0_pp=zero; real_time=zero
!----- time stepping loop -----
IF(numpe==it) THEN
  WRITE(11,'(A)') " Time t cos(omega*t) Displacement Iterations"
END IF
timesteps: DO j=1,nstep
  real_time=real_time+dtim; loads_pp=zero; u_pp=zero; vu_pp=zero
  elements_3: DO iel=1,nels_pp      ! gather for rhs multiply
    temp_pp(:,:,iel)=store_km_pp(:,:,iel)*c2+store_mm_pp(:,:,iel)*c3
  END DO elements_3; CALL gather(x0_pp,pmul_pp)
  DO iel=1,nels_pp
    utemp_pp(:,iel)=MATMUL(temp_pp(:,:,iel),pmul_pp(:,iel))
  END DO; CALL scatter(u_pp,utemp_pp)
!----- velocity part -----
temp_pp=store_mm_pp/theta; CALL gather(d1x0_pp,pmul_pp)
DO iel=1,nels_pp
  utemp_pp(:,iel)=MATMUL(temp_pp(:,:,iel),pmul_pp(:,iel))
END DO; CALL scatter(vu_pp,utemp_pp)      ! doesn't add to last u_pp
loads_pp=fext_pp*(theta*dtim*cos(omega*real_time)+c1*          &
                 cos(omega*(real_time-dtim)))
loads_pp=u_pp+vu_pp+loads_pp
!----- solve simultaneous equations by PCG -----
d_pp=diag_precon_pp*loads_pp; p_pp=d_pp; x_pp=zero; iters=0
iterations: DO
  iters=iters+1; u_pp=zero; vu_pp=zero
  temp_pp=store_mm_pp*c3+store_km_pp*c4
  CALL gather(p_pp,pmul_pp)
  elements_4: DO iel=1,nels_pp
    utemp_pp(:,iel)=MATMUL(temp_pp(:,:,iel),pmul_pp(:,iel))
  END DO elements_4; CALL scatter(u_pp,utemp_pp)
  up=DOT_PRODUCT_P(loads_pp,d_pp); alpha=up/DOT_PRODUCT_P(p_pp,u_pp)
  xnew_pp=x_pp+p_pp*alpha; loads_pp=loads_pp-u_pp*alpha
  d_pp=diag_precon_pp*loads_pp; beta=DOT_PRODUCT_P(loads_pp,d_pp)/up
  p_pp=d_pp+p_pp*beta; u_pp=xnew_pp
  CALL checon_par(xnew_pp,tol,converged,x_pp)
  IF(converged.OR.iters==limit) EXIT
END DO iterations
x1_pp=xnew_pp
d1x1_pp=(x1_pp-x0_pp)/(theta*dtim)-d1x0_pp*(1._iwp-theta)/theta
d2x1_pp=(d1x1_pp-d1x0_pp)/(theta*dtim)-d2x0_pp*(1._iwp-theta)/theta
x0_pp=x1_pp; d1x0_pp=d1x1_pp; d2x0_pp=d2x1_pp; utemp_pp=zero
IF(j/npri*npri==j) THEN
  IF(numpe==it) WRITE(11,'(3E12.4,I10)') real_time,          &
    cos(omega*real_time),x1_pp(is),iters
  IF(numpe==1) THEN; WRITE(ch,'(I6.6)') j
  OPEN(12,file=argv(1:nlen)//".ensi.DISPL-//ch,status='replace', &
        action='write'); WRITE(12,'(A)')          &
    "Alya Ensight Gold --- Vector per-node variable file"
  WRITE(12,'(A/A/A)') "part", "1", "coordinates"
END IF

```

```

CALL gather(x1_pp(1:),eld_pp); disp_pp=zero
CALL scatter_nodes(npes,mn,nelss_pp,g_num_pp,nod,ndim,nodes_pp,
    node_start,node_end,eld_pp,disp_pp,1) &
DO i=1,ndim ; temp=zero
    DO l=1,nodes_pp; k=i+(ndim*(l-1)); temp(l)=disp_pp(k); END DO
    CALL dismsh_ensi_p(12,l,nodes_pp,npes,numpe,1,temp)
END DO ; IF(numpe==1) CLOSE(12)
END IF
END DO timesteps
IF(numpe==it) THEN
    WRITE(11,'(A,F10.4)') "This analysis took:", elap_time()-timest(1)
END IF; CALL SHUTDOWN()
END PROGRAM p129

```

The now familiar consistency of loops `elements_1` and `elements_2` with embedded `gauss_pts_1` appears again. The parallel (3D) version has the lumped mass matrix for 20-node elements hand-coded in place of the `elmat` routine used in the serial (2D) case which used 8-node elements. The time-stepping loop involves comparable loop `elements_3` while the `pcg` section involves comparable loop `elements_4`. Data are listed as Figure 12.47 with results as Figure 12.48. Performance statistics are listed in Figure 12.49 for this problem and in Figure 12.50 for a more refined mesh.

```

program
'p129'

iotype      nels nxr nze nip
'parafem'  2560   8   8   27

aa          bb          cc          rho
0.125     0.125     0.125     2000.0

e           v
10000.0    0.3

alpha1      beta1
0.0008     0.5

nstep      npri theta
32         1     1.0

omega      tol      limit
0.01       0.0001   3000

```

**Figure 12.47** Data for Program 12.9 example

```

This job ran on      16 processes
There are 12465 nodes 225 restrained and 36720 equations
Time after setup was: 0.0980
      Time t      cos(omega*t) Displacement Iterations
      0.3142E+02  0.9511E+00 -0.1436E-03      422
      0.6283E+02  0.8090E+00 -0.1275E-03      422
      0.9425E+02  0.5878E+00 -0.9295E-04      458
      .
      .
      0.9425E+03  -0.1000E+01  0.1517E-03      423
      0.9739E+03  -0.9511E+00  0.1524E-03      427
      0.1005E+04  -0.8090E+00  0.1277E-03      461
This analysis took : 85.0621

```

**Figure 12.48** Results from Program 12.9 example

Mesh	No of MPI Processes	Analysis Time (secs)
8x40x8	16	85
	32	43

**Figure 12.49** Performance statistics: Program 12.9 (Cray XE6)

5M elements, 61M equations

Mesh	No of MPI Processes	Analysis Time (secs)
100x500x100	256	28,879
	512	14,920
	1024	7,803
	2048	4,318

**Figure 12.50** Larger data set: Program 12.9 (Cray XE6)

## Program 12.10 Forced vibration analysis of three-dimensional elasto plastic solid. Explicit integration in time. Compare Program 11.8

```

PROGRAM p1210
!-----
! Program 12.10 Forced vibration of an elastic-plastic(Von Mises) solid.
!                         Viscoplastic strain method, lumped mass, explicit
!                         integration
!-----
!USE mpi_wrapper !remove comment for serial compilation
USE precision; USE global_variables; USE mp_interface; USE input
USE output; USE loading; USE timing; USE maths; USE gather_scatter
USE new_library; IMPLICIT NONE
! neq,ntot are now global variables - not declared
INTEGER,PARAMETER::nodof=3,ndim=3,nst=6
INTEGER::nn, nr, nip, loaded_nodes, nres=1, nod, i, j, k, ii, jj, iel, nstep, npri, &
is, it, nlen, ndof, nels, npes_pp, node_end, node_start, nodes_pp, meshgen, &
partitioner
REAL(iwp)::rho, dtim, e, v, det, sbary, sigm, f, fnew, fac, volume, sbar, &
dsbar, lode_theta, real_time, tload, pload
REAL(iwp),PARAMETER::zero=0.0_iwp
CHARACTER(LEN=15)::element; CHARACTER(LEN=50)::argv; CHARACTER(LEN=6)::ch
!----- dynamic arrays -----
REAL(iwp),ALLOCATABLE::points(:,:), bdylds_pp(:), x1_pp(:), d1x1_pp(:), &
stressv(:,:,:), pl(:,:,:), emm(:), d2x1_pp(:), tensor_pp(:,:,:,:), &
etensor_pp(:,:,:,:), val(:,:,:), dee(:,:,:), mm_pp(:), jac(:,:), weights(:), &
der(:,:), deriv(:,:), bee(:,:), eld(:), eps(:), sigma(:), bload(:), &
eload(:), mm_tmp(:,:), g_coord_pp(:,:,:,:), pmul_pp(:,:,:), utemp_pp(:,:,:), &
disp_pp(:), fext_pp(:), temp(:)
INTEGER,ALLOCATABLE::timest(:, ), rest(:, :, ), no(:, ), node(:, ), g_num_pp(:, :, ), &
g_g_pp(:, : )
!----- input and initialisation -----
ALLOCATE(timest(20)); timest=zero; timest(1)=elap_time()
CALL find_pe_procs(numpe,npes); CALL getname(argv,nlen)
CALL read_p1210(argv,numpe,dtim,e,element,loaded_nodes,meshgen,nels, &
nip,nn,nod,npri, nr, nres,nstep,partitioner,pload,rho,sbary,v)
CALL calc_nels_pp(argv, nels, npes, numpe, partitioner, nels_pp)

```

```

ndof=nod*nodof; ntot=ndof
ALLOCATE(g_num_pp(nod,nels_pp),g_coord_pp(nod,ndim,nels_pp),
         rest(nr,nodof+1)); g_num_pp=0; g_coord_pp=zero; rest=0
CALL read_g_num_pp(argv,iel_start,nn,npes,numpe,g_num_pp)
IF(meshgen == 2) THEN
    CALL abaqus2sg(element,g_num_pp)
END IF
CALL read_g_coord_pp(argv,g_num_pp,nn,npes,numpe,g_coord_pp)
CALL read_rest(argv,numpe,rest)
ALLOCATE(points(nip,ndim),weights(nip),dee(nst,nst),
        pmul_pp(ntot,nels_pp),tensor_pp(nst,nip,nels_pp),no.loaded_nodes),
        &
        pl(nst,nst),etensor_pp(nst,nip,nels_pp),jac(ndim,ndim),der(ndim,nod),
        &
        deriv(ndim,nod),bee(nst,ntot),eld(ntot),eps(nst),sigma(nst),emm(ntot),
        &
        bload(ntot),eload(ntot),stressv(nst),g_g_pp(ntot,nels_pp),
        &
        mm_tmp(ntot,nels_pp),utemp_pp(ntot,nels_pp))
!----- find the steering array and equations per process -----
CALL rearrange(rest); g_g_pp=0; neq=0
elements_0: DO iel=1,nels_pp
    CALL find_g(g_num_pp(:,iel),g_g_pp(:,iel),rest)
END DO elements_0
neq=MAXVAL(g_g_pp); neq=max_p(neq); CALL calc_neq_pp
CALL calc_npes_pp(npes,npes_pp); CALL make_ggl(npes_pp,npes,g_g_pp)
DO i=1,neq_pp; IF(nres==ieq_start+i-1)THEN;it=numpe;is=i;END IF;END DO
IF(numpe==it) THEN
    OPEN(11,FILE=argv(1:nlen)//".res",STATUS='REPLACE',ACTION='WRITE')
    WRITE(11,'(A,I6,A)') "This job ran on ",npes," processes"
    WRITE(11,'(A,3(I12,A))') "There are ",nn," nodes ",nr,
        &
        " restrained and ", neq, " equations"
    WRITE(11,'(A,F10.4)') "Time after setup was:", elap_time()-timest(1)
END IF
CALL calc_nodes_pp(nn,npes,numpe,node_end,node_start,nodes_pp)
ALLOCATE(disp_pp(nodes_pp*ndim),temp(nodes_pp)); disp_pp=zero; temp=0
ALLO-
CATE(bdylds_pp(neq_pp),x1_pp(neq_pp),d1x1_pp(neq_pp),mm_pp(neq_pp), &
      d2x1_pp(neq_pp),fext_pp(neq_pp)); bdylds_pp=zero; x1_pp=zero
      d1x1_pp=zero; d2x1_pp=zero; mm_pp=zero; fext_pp=zero
!----- calculate diagonal mass matrix -----
mm_tmp=zero; CALL sample(element,points,weights)
elements_1: DO iel=1,nels_pp
    volume=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); jac=MATMUL(der,g_coord_pp(:,:,iel))
        det=determinant(jac); volume=volume+det*weights(i)*rho
    END DO gauss_pts_1
    emm=volume/13._iwp; emm(1:19:6)=emm(4)*.125_iwp
    emm(2:20:6)=emm(4)*.125_iwp; emm(3:21:6)=emm(4)*.125_iwp
    emm(37:55:6)=emm(4)*.125_iwp; emm(38:56:6)=emm(4)*.125_iwp
    emm(39:57:6)=emm(4)*.125_iwp; mm_tmp(:,iel)=mm_tmp(:,iel)+emm
END DO elements_1
CALL scatter(mm_pp,mm_tmp); DEALLOCATE(mm_tmp)
!----- loads -----
IF(loaded_nodes>0) THEN
    ALLOCATE(node.loaded_nodes), val(ndim,loaded_nodes)); val=zero; node=0
    CALL read_loads(argv,numpe,node,val)
    CALL load(g_g_pp,g_num_pp,node,val,fext_pp(1:))
    tload = SUM_P(fext_pp(1:)); DEALLOCATE(node,val)
END IF
!----- explicit integration loop -----

```

```

tensor_pp=zero; etensor_pp=zero; real_time=zero
IF(numpe==it) THEN
    WRITE(11,'(A)') " Time      Displacement Velocity Acceleration "
    WRITE(11,'(4E12.4)') real_time,x1_pp(is),d1x1_pp(is),d2x1_pp(is)
END IF
time_steps: DO jj=1,nstep
    real_time=real_time+dtime; bdylds_pp=zero
    x1_pp=x1_pp+(d1x1_pp+d2x1_pp*dtime*.5_iwp)*dtime
!----- element stress-strain relationship -----
    pmul_pp=zero; utemp_pp=zero; CALL gather(x1_pp,pmul_pp)
elements_2: DO iel=1,nels_pp
    bload=zero; eld=pmul_pp(:,iel)
    gauss_pts_2: DO i=1,nip
        dee=zero; CALL deemat(dee,e,v); CALL shape_der(der,points,i)
        jac=MATMUL(der,g_coord_pp(:,:,iel)); det=determinant(jac)
        CALL invert(jac); deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        eps=MATMUL(bee,pmul_pp(:,iel)); eps=eps-etenso_pp(:,i,iel)
        sigma=MATMUL(dee,eps); stressv=sigma+tensor_pp(:,i,iel)
        CALL invar(stressv,sigm,dsbar,lode_theta); fnew=dsbar-sbary
!----- check whether yield is violated -----
        IF(fnew>=.0_iwp)THEN
            stressv=tensor_pp(:,i,iel)
            CALL invar(stressv,sigm,sbar,lode_theta)
            f=sbar-sbary; fac=fnew/(fnew-f)
            stressv=tensor_pp(:,i,iel)+(1._iwp-fac)*sigma
            CALL vmp1(e,v,stressv,p1); dee=dee-fac*p1
        END IF
        sigma=MATMUL(dee,eps); sigma=sigma+tensor_pp(:,i,iel)
        eload=MATMUL(sigma,bee); bload=bload+eload*det*weights(i)
!----- update the gauss points -----
        tensor_pp(:,i,iel)=sigma
        etensor_pp(:,i,iel)=etenso_pp(:,i,iel)+eps
        END DO gauss_pts_2; utemp_pp(:,iel)=utemp_pp(:,iel)-bload
    END DO elements_2
    CALL scatter(bdylds_pp,utemp_pp); bdylds_pp=bdylds_pp+fext_pp*pload
    bdylds_pp=bdylds_pp/mm_pp
    d1x1_pp=d1x1_pp+(d2x1_pp+bdylds_pp)*.5_iwp*dtime; d2x1_pp=bdylds_pp
!----- output displacements -----
    IF(jj==jj/npri) THEN
        IF(numpe==it) THEN
            WRITE(11,'(4E12.4)') real_time,x1_pp(is),d1x1_pp(is),d2x1_pp(is)
        END IF
        IF(numpe==1) THEN; WRITE(ch,'(I6.6)') jj
            OPEN(12,file=argv(1:nlen)//".ensi.DISPL"//ch,status='replace', &
                  action='write'); WRITE(12,'(A)') &
            "Alya Ensign Gold --- Vector per-node variable file"
            WRITE(12,'(A/A/A)') "part", " 1", "coordinates"
        END IF; disp_pp=zero; utemp_pp=zero
        CALL gather(x1_pp(1:),utemp_pp)
        CALL scatter_nodes(npes,nn,nels_pp,g_num_pp,nod,ndim,nodes_pp, &
                          node_start,node_end,utemp_pp,disp_pp,1)
        DO i=1,ndim ; temp=zero
            DO ii=1,nodes_pp; k=i+(ndim*(ii-1)); temp(ii)=disp_pp(k); END DO
            CALL dismsh_ensi_p(12,jj,nodes_pp,npes,numpe,1,temp)
        END DO ; IF(numpe==1) CLOSE(12)
    END IF
END DO time_steps
IF(numpe==it) THEN

```

```

      WRITE(11,'(A,F10.4)')"This analysis took:",elap_time()-timest(1)
      END IF; CALL SHUTDOWN()
END PROGRAM p1210

```

In the parallel version, an extra loop, `elements_0`, creates the distributed steering array `g_g_pp` as is done by `elements_1` in serial. Then, parallel `elements_1` creates the lumped mass matrix for 20-node hexahedral elements. Following this, loops `elements_2` in both programs are equivalent. Uniform elements are assumed in 3D. Data are listed as Figure 12.51 with results as Figure 12.52. Performance statistics for this problem and a refined mesh are provided in Figures 12.53 and 12.54, respectively. The small problem was run on an SGI Intel Cluster and the large problem on a Cray XE6.

```

program
'p1210'

iotype     nels nxe nze nip
'parafem' 5000  10  10  27

aa      bb      cc
0.1    0.1    0.1

e       v       rho   sbary
1.0    1.0    0.3   1.0E8

dtim    nstep   npri
0.005   1000    100

```

**Figure 12.51** Data for Program 12.10 example

```

This job ran on      32 processes
There are        23441 nodes          341 restrained and      69300 equations
Time after setup was: 1.1118
      Time      Displacement      Velocity      Acceleration
0.0000E+00  0.0000E+00  0.0000E+00  0.0000E+00
0.5000E+00  0.3679E-03 -0.1106E-01 -0.1647E+01
0.1000E+01 -0.2494E-04  0.1781E-01  0.3397E+00
0.1500E+01  0.6773E-03  0.3175E-02  0.1026E+01
0.2000E+01  0.5629E-02  0.2656E-01  0.3073E+00
0.2500E+01  0.4803E-02  0.9779E-02  0.4200E+00
0.3000E+01  0.1071E-01  0.1293E-01 -0.5167E-01
0.3500E+01  0.9944E-02 -0.6045E-02 -0.7788E-01
0.4000E+01  0.1761E-01  0.2178E-01 -0.8384E+00
0.4500E+01  0.1872E-01  0.1129E-01  0.2836E+00
0.5000E+01  0.1950E-01  0.3994E-01  0.2788E+00
This analysis took: 21.7857

```

**Figure 12.52** Results from Program 12.10 example

Mesh	No of MPI Processes	Analysis Time (secs)
10x50x10	8	60
	16	37
	32	22

**Figure 12.53** Performance statistics: Program 12.10

625,000 elements, 7.7M equations		
Mesh	No of MPI Processes	Analysis Time (secs)
50x250x50	128	2,074
	256	1,075
	512	561
	1024	310

**Figure 12.54** Larger data set: Program 12.10

## 12.3 Graphics Processing Units

GPUs were introduced in Section 1.6. Here, Program 12.11 shows how to use GPUs via compiler directives. At the time of writing, compiler directives for GPUs are only available with specialist compilers. Here we use the CAPS compiler with OpenHMPP directives ([www.caps-entreprise.com](http://www.caps-entreprise.com)). The benefit of this strategy is that if there is no GPU hardware available or the user does not have access to a compiler that supports GPU directives, the resulting source code can still be compiled for a standard CPU. ‘Normal’ compilers treat the directives as comments and ignore them.

### 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

```
PROGRAM p1211
!-----
! Program 12.11 Three-dimensional strain of an elastic solid using
!           8-, 14- or 20-node brick hexahedra. Mesh numbered in x-y
!           planes then in the z-direction. No global stiffness matrix
!           assembly. Diagonally preconditioned conjugate gradient
!           solver. GPU version.
!-----
USE new_library; USE geometry; USE precision; USE timing
USE maths; USE global_variables; USE input; IMPLICIT NONE
!INCLUDE 'altcublas.inc' ! Include HMPPALT cublas proxy interface
!neq is global, must not be declared
INTEGER::err ! For HMPPALT error management
REAL(iwp)::coeffa=1_iwp, coeffb=0_iwp ! Coefficients for DGEMM
INTEGER::cg_iters,cg_limit,fixed_freedoms,i,iel,k,loaded_nodes,ndim=3,   &
ndof,nels,nip,nn,nprops=2,np_types,nod,nodof=3,nr,nst=6,nxe,nye,nze,nlen
REAL(iwp)::alpha,beta,big,cg_tol,det,one=1.0_iwp,penalty=1.0e20_iwp,up,   &
zero=0.0_iwp
CHARACTER(LEN=15)::element='hexahedron',argv
LOGICAL::cg_converged
!-----dynamic arrays-----
INTEGER,ALLOCATABLE::etype(:),g(:,g_g(:,:,),g_num(:,:,),nf(:,:,),no(:),      &
node(:),num(:),sense(:)
REAL(iwp),ALLOCATABLE::bee(:,:,),coord(:,:,),d(:,:,),dee(:,:,),der(:,:,),      &
deriv(:,:),diag_precon(:),eld(:),fun(:),gc(:,g_coord(:,:,),g_pmul(:,:,),&
g_utemp(:,:)),jac(:,:,),km(:,:,),loads(:,p(:),points(:,:,),prop(:,:,),      &
sigma(:,store(:,storkm(:,:,),u(:,value(:,weights(:,x(:,xnew(:,&
x_coords(:,y_coords(:,z_coords(:,timest(:,oldlds(:)
```

```

!-----external subroutines-----
EXTERNAL::DDOT ; REAL(iwp)::DDOT ! For MKL
!-----input and initialisation-----
ALLOCATE(timest(2)); timest=zero; timest(1) = elap_time()
CALL getname(argv,nlen)
OPEN(10,FILE=argv(1:nlen)//'.dat'); OPEN(11,FILE=argv(1:nlen)//'.res')
READ(10,*)nod,nxe,nye,nze,nip,cg_tol,cg_limit,np_types
CALL mesh_size(element,nod,nels,nn,nxe,nye,nze); ndof=nod*nodof
ALLOCATE(nf(nodof,nn),points(nip,ndim),dee(nst,nst),coord(nod,ndim),
jac(ndim,ndim),der(ndim,nod),deriv(ndim,nod),fun(nod),gc(ndim),
bee(nst,ndof),km(ndof,ndof),eld(ndof),sigma(nst),g_coord(ndim,nn),
g_num(nod,nels),weights(nip),num(nod),g_g(ndof,nels),x_coords(nxe+1),
g(ndof),y_coords(nye+1),z_coords(nze+1),etype(nels),g_pmul(ndof,nels),
prop(nprops,np_types),storkm(ndof,nodof,nels),g_utemp(ndof,nels))
READ(10,*)
prop; etype=1; IF(np_types>1)READ(10,*)etype
READ(10,*)x_coords,y_coords,z_coords
nf=1; READ(10,*)nr,(k,nf(:,k),i=1,nr); CALL formmf(nf); neq=MAXVAL(nf)
ALLOCATE(p(0:neq),loads(0:neq),x(0:neq),xnew(0:neq),u(0:neq),
diag_precon(0:neq),d(0:neq),oldlds(0:neq))
diag_precon=zero; CALL sample(element,points,weights)
timest(2)=elap_time()

!-----element stiffness integration, storage and preconditioner-----
elements_1: DO iel=1,nels
    CALL hexahedron_xz(iel,x_coords,y_coords,z_coords,coord,num)
    CALL num_to_g(num,nf,g); g_num(:,iel)=num
    g_coord(:,num)=TRANSPOSE(coord); g_g(:,iel)=g
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)));num=g_num(:,iel)
    coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel); km=zero
    gauss_pts_1: DO i=1,nip
        CALL shape_der(der,points,i); jac=MATMUL(der,coord)
        det=determinant(jac); CALL invert(jac)
        deriv=MATMUL(jac,der); CALL beemat(bee,deriv)
        km=km+MATMUL(matmul(transpose(bee),dee),bee)*det*weights(i)
    END DO gauss_pts_1
    storkm(:,:,iel) = km
    DO k=1,ndof; diag_precon(g(k))=diag_precon(g(k))+km(k,k); END DO
END DO elements_1

!-----invert the preconditioner and get starting loads--
loads=zeros; READ(10,*)loaded_nodes,(k,loads(nf(:,k)),i=1,loaded_nodes)
oldlds=loads; READ(10,*)fixed Freedoms
IF(fixed Freedoms/=0)THEN
    ALLOCATE(node(fixed Freedoms),sense(fixed Freedoms),
value(fixed Freedoms),no(fixed Freedoms),store(fixed Freedoms)) &
    READ(10,*)(node(i),sense(i),value(i),i=1,fixed Freedoms)
    DO i=1,fixed Freedoms; no(i)=nf(sense(i),node(i)); END DO
    diag_precon(no)=diag_precon(no)+penalty; loads(no)=diag_precon(no)*value
    store=diag_precon(no)
END IF
diag_precon(1)=one/diag_precon(1:); diag_precon(0)=zero
d=diag_precon*loads; p=d

!-----pcg equation solution-----
!$hmp htspt1 acquire
!$hmp htspt1 region, target=CUDA
!$hmp htspt1 endregion
x=zero; cg_iters=0
!$hmp htspt1 allocate data["km"], size={ndof,ndof}
!$hmp htspt1 allocate data["g_pmul"], size={ndof,nels}
!$hmp htspt1 allocate data["g_utemp"], size={ndof,nels}

```

```

!$hmpp htspt1 advancedload data["km"], asynchronous
pcg: DO
    cg_iters=cg_iters+1; u=zero; g_utemp=zero
    elements_2: DO iel=1,nels
        g_pmul(:,iel)=p(g_g(:,iel))
    END DO elements_2
    !$hmpp htspt1 advancedload data["g_pmul"]
    !$hmpp htspt1 waitload data["km"]
    !$hmppalt cublas call, name="dgemm", error="err"
    CALL DGEMM('N','N',ndof,nels,ndof,coeffa,km,ndof,g_pmul,ndof,coeffb,  &
               g_utemp,ndof)
    !$hmpp htspt1 delegatedstore data["g_utemp"]
    elements_2a: DO iel=1,nels
        u(g_g(:,iel)) = u(g_g(:,iel))+g_utemp(:,iel)
    END DO elements_2a
    IF(fixed_freedoms/=0) u(no)=p(no)*store
    up=DDOT(neq,loads,1,d,1); alpha=up/DDOT(neq,p,1,u,1)
    xnew=x+p*alpha; loads=loads-u*alpha; d=diag_precon*loads
    beta=DDOT(neq,loads,1,d,1)/up; p=d+p*beta
    CALL checon(xnew(1:),x(1:),cg_tol,cg_converged)
    IF(cg_converged.OR.cg_iters==cg_limit) EXIT
END DO pcg
!$hmpp htspt1 release
WRITE(11,'(A)')                                     &
"This analysis ran on 1 process and was accelerated using 1 GPU"
WRITE(11,'(A,I8,A)') "There are",neq," equations"
WRITE(11,'(A,I5)') "Number of iterations to convergence was ",cg_iters
WRITE(11,'(/A)')          Node      x-disp      y-disp      z-disp"
DO k=1,nn,nn-1; WRITE(11,'(I10,3E12.4)')k,xnew(nf(:,k)); END DO
!----- recover stresses at nip integrating point -----
nip      = 1; DEALLOCATE(points,weights)
ALLOCATE(points(nip,ndim),weights(nip))
CALL sample(element,points,weights)
WRITE(11,'(/A,I2,A)')"The integration point (nip=",nip,") stresses are:"
WRITE(11,'(A,,A)')"      Element      x-coord      y-coord      z-coord",  &
"      sig_x      sig_y      sig_z      tau_xy      tau_yz      tau_zx"
xnew(0)=zero
elements_3: DO iel=1,nels
    CALL deemat(dee,prop(1,etype(iel)),prop(2,etype(iel)))
    num=g_num(:,iel); coord=TRANSPOSE(g_coord(:,num)); g=g_g(:,iel)
    eld=xnew(g)
    gauss_pts_2: DO i=1,nip
        CALL shape_der(der,points,i); CALL shape_fun(fun,points,i)
        gc=MATMUL(fun,coord); jac=MATMUL(der,coord)
        CALL invert(jac); deriv=MATMUL(jac,der)
        CALL beemat(bee,deriv); sigma=MATMUL(dee,MATMUL(bee,eld))
        IF(iel<4) THEN
            WRITE(11,'(I8,4X,3E12.4)')iel,gc; WRITE(11,'(6E12.4)')sigma
        END IF
    END DO gauss_pts_2
END DO elements_3
WRITE(11,'(/A,F12.4,A)') "Time for setup was           ",  &
                           timest(2)-timest(1),"s"
WRITE(11,'(A,F12.4,A)') "Total time for this analysis was      ",  &
                           elap_time()-timest(1),"s"
STOP
END PROGRAM p1211

```

Program 12.11 is essentially the same as Program 5.7 and is the only program in this chapter that does not use MPI. Instead, parallel processing is achieved using hundreds of cores available on the GPU. The main hotspot, which was identified in Chapter 5, is the multiplication of `km` and `g_pmul` in the preconditioned conjugate gradient solver. This is therefore the first target for acceleration using the GPU.

The first change to note is the addition of the statement `INCLUDE 'altcublas.inc'` at the declaration stage of the program. This refers the compiler to an external file `altcublas.inc` which provides a FORTRAN interface between the program and the external CUBLAS maths library. CUBLAS is written (for NVIDIA GPUs only) in the C programming language.

A number of compiler directives appear later in the program. These start with the text `!$hmp` followed by the required action. Program sections targeted for GPU acceleration are usually enclosed by the directives `region` and `end region`. A label `htspt1` associates these sections with GPU operations such as memory allocations. Here the region is empty because the only section to be off-loaded to the GPU is the BLAS routine `DGEMM`. The directive `!$hmpalt cublas` call instructs the compiler to replace `DGEMM` with the equivalent call to CUBLAS.

Before any computation can be carried out on the GPU, the computer operating system should allow the program to access the GPU hardware through the directive `$hmp htspt1 acquire`. After this step, data needs to be transferred from the host CPU memory to the on-board GPU memory. Memory is allocated on the GPU using a separate call to `!$hmp htspt1 allocate` for each data item. The matrix–matrix multiplication multiplies `km` and `g_pmul`, returning the result in `g_utemp`. Transfer of `km` and `g_pmul` from the host to the GPU is carried out by `!$hmp htspt1 advanced-load` and transfer of `g_utemp` from the GPU back to the host requires the directive `!$hmp htspt1 delegatedstore`.

The benefit of using GPUs to accelerate numerical computations is often limited by the time it takes to transfer data between the GPU and host. Here, data transfer for large arrays (dimensioned by `NELS`) is required for each iteration of the solver. There is no opportunity to overlap communications with computation as the matrix–matrix multiplication is sandwiched between two blocks of data transfer.

Timings for a problem similar to Figure 5.41 in Chapter 5, comprising 200,000 20-node hexahedra, are presented in Table 12.7. Data for the problem is shown in Figure 12.55 and results as Figure 12.56. The CPU sequential times were recorded using the GFORTRAN compiler and `DGEMM` on an 8-core Intel Xeon E5 processor. The CPU parallel times made use of a parallel version of Intel’s `DGEMM` subroutine (from the INTEL MKL library) on

**Table 12.7** Comparison of CPU and GPU run times

	CPU sequential	CPU parallel	GPU parallel
Total run time (s)	977	158	205
Equation solution using <code>pcg</code> (s)	928	131	177
GPU initialisation (s)	—	—	2
Data transfer (s)	—	—	73
<code>DGEMM</code> (s)	718	40	0.66
Speed-up of <code>pcg</code> vs. CPU sequential	1	7	5.3

```

nod
20

nx e  ny e  nze  nip  cg_tol    cg_limit  np_types
32     96     64     8   1.00E-05   2000      1

prop(e,v)
100  0.3

etyp e
not used

x_coords, y_coords, z_coords
0.0000  0.0156  0.0313  0.0469  0.0625  0.0781  0.0938  0.1094  0.1250
0.1406  0.1563  0.1719  0.1875  0.2031  0.2188  0.2344  0.2500  0.2656
.
.
.
-1.6875 -1.7188 -1.7500 -1.7813 -1.8125 -1.8438 -1.8750 -1.9063 -1.9375
-1.9688 -2.0000

nr, (k,nf(:,k),i=1,nr)
34177
    1 0 0 1      2 1 0 1      3 1 0 1      4 1 0 1      5 1 0 1
    6 1 0 1      7 1 0 1      8 1 0 1      9 1 0 1      10 1 0 1
.
.
.
820598 0 0 0  820599 0 0 0  820600 0 0 0  820601 0 0 0  820602 0 0 0
820603 0 0 0  820604 0 0 0  820605 0 0 0  820606 0 0 0  820607 0 0 0
820608 0 0 0  820609 0 0 0

loaded_nodes, (k,loadsUnf(:,k)),i=1,loaded_nodes
3201
    1  0.00000000E+00  0.00000000E+00  0.40690104E-04
    2  0.00000000E+00  0.00000000E+00  -0.16276042E-03
    3  0.00000000E+00  0.00000000E+00  0.81380208E-04
.
.
.
271487  0.00000000E+00  0.00000000E+00  0.81380208E-04
271488  0.00000000E+00  0.00000000E+00  -0.16276042E-03
271489  0.00000000E+00  0.00000000E+00  0.40690104E-04

fixed_freedoms
0

```

**Figure 12.55** Data for Program 12.11 example

the same processor. The GPU times used the HMPP compiler, CUBLAS and an Nvidia Tesla K20 GPU board.

The results are striking. DGEMM runs more than 1000 times faster in parallel on the GPU than sequentially on the CPU. However, the long data transfer times, back and forth between the host CPU and GPU at every iteration (adding a total of 73 s data transfer to 0.66 s of computation), means that the best overall implementation is ‘CPU parallel’, which uses DGEMM in parallel on the CPU (compare total run times). The Intel MKL version of DGEMM makes good use of the eight cores available on the processor and runs seven times faster than the sequential version of DGEMM.

Attempts were made to improve the GPU-enabled program by moving the elements\_2 and elements\_2a loops to the GPU, thus removing the need to transfer data. The benefits gained were outweighed by the random way in which the global steering vector g\_g accesses memory as it scatters data to the vector u. This is

This analysis ran on 1 process and was accelerated using 1 GPU  
 There are 2408576 equations  
 Number of iterations to convergence was 1247

Node	x-disp	y-disp	z-disp
1	0.0000E+00	0.0000E+00	-0.1620E-01
820609	0.0000E+00	0.0000E+00	0.0000E+00

The integration point (nip= 1) stresses are:

Element	x-coord	y-coord	z-coord	sig_x	sig_y	sig_z	tau_xy	tau_yz	tau_zx
1	0.7800E-02	0.1565E-01	-0.1565E-01	0.5518E-02	-0.5998E+00	-0.9991E+00	0.1408E-04	-0.2968E-04	0.2067E-04
2	0.2345E-01	0.1565E-01	-0.1565E-01	0.1077E-02	-0.6015E+00	-0.1001E+01	0.4898E-04	-0.2031E-04	0.5654E-04
3	0.3910E-01	0.1565E-01	-0.1565E-01	0.5484E-02	-0.5995E+00	-0.9991E+00	0.8264E-04	-0.2297E-04	0.7797E-04

Time for setup was 0.0530s  
 Total time for this analysis was 204.5076s

**Figure 12.56** Results from Program 12.11 example

difficult to parallelise. Similar issues were found in the MPI programs and the solution was to build more efficient communication tables using the subroutine makeggl.

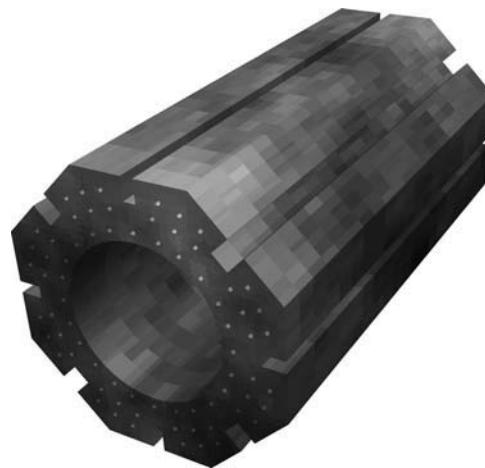
Program 12.11 is the best GPU version that could be achieved without rewriting the original FORTRAN code. The lesson learnt is that GPUs offer the potential to significantly accelerate finite element computations if the bottlenecks described here are removed. Furthermore, it should also be noted that the next step, using multiple GPUs at the same time, can be achieved by adopting the MPI strategy presented earlier in this chapter, whereby each GPU would operate on its own set of elements. Finally, the use of compiler directives means that the program is portable between platforms that do or do not have GPU hardware available.

## 12.4 Cloud Computing

Use of Programs 12.1 to 12.10 for solving ‘large’ problems assumes access to a supercomputer. Whilst these are becoming more widely available in research organisations and large companies, they are still out of reach for small and medium-sized firms. An alternative to owning a supercomputer is to hire somebody else’s system on a ‘pay as you go’ basis (Cloud Computing). At the time of writing, a number of firms such as SGI provide access to supercomputers similar to those used for performance evaluation in this chapter.

The most common use of Cloud Computing is to carry out computations that are embarrassingly parallel and require no inter-process communication. The main companies offering this type of service are global firms such as Amazon, Google and Microsoft. Amazon and Google own vast ‘server farms’ which have a capacity designed to cope with surges in demand, for online shopping and online searches, respectively. Cloud Computing is therefore a way of generating additional revenue from their ‘excess’ capacity.

This section describes the use of the Microsoft service ‘Azure’ to investigate whether the mechanical behaviour of nuclear graphite bricks is sensitive to tiny variations in elastic stiffness. Program 5.6 is used together with SIM3DE, a random field generator from Fenton and Griffiths (2008). Experimentally derived values for the mean Young’s modulus of graphite, together with the standard deviation, are used to generate a ‘random field’ of property values in the finite element model. Figure 12.57 shows the geometry of



**Figure 12.57** Nuclear graphite brick example

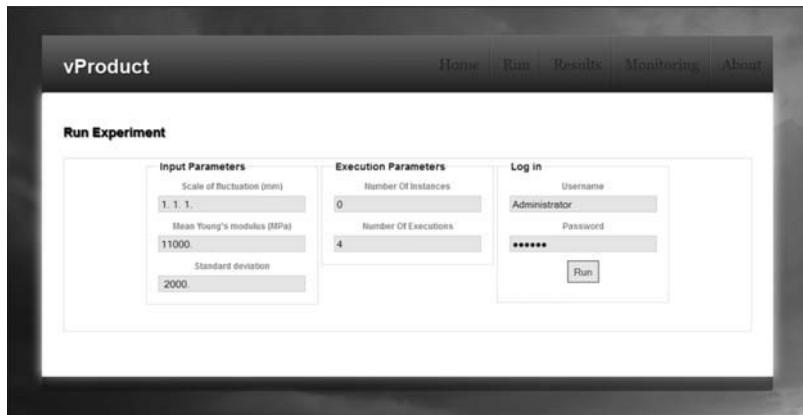
the brick and a ‘realisation’ of property values ranging from less stiff (light) to more stiff (dark) than the mean. The full study requires repeating the analysis hundreds of times with different, randomly generated, distributions of Young’s modulus. These could be carried out one after the other on a desktop computer or all at the same time on Azure.

Running the book programs on Microsoft Azure requires significant effort and only brief details are given here. A full explanation is given by Christias (2012). To develop their own Microsoft Azure ‘deployment’, the interested reader is also recommended to consult step-by-step tutorials provided by Azure (<http://www.windowsazure.com>).

On the developer side, Microsoft Azure provides a number of tools that are downloaded onto the desktop PC in order to build a Cloud application. The FORTRAN programs are written and compiled as normal. However, to run on Microsoft Azure, they need to be ‘wrapped’ by special code written in C#. This sounds onerous, however the C# code is provided by Microsoft Azure and only requires minor edits. An online control panel is used to launch Microsoft Azure storage and compute services. Cloud storage is required for the model input data, the executable and the results of the simulations. Once the storage service has been started, the input data and wrapped executable can be uploaded to Microsoft Azure from the desktop via a data transfer tool. A set of software ‘components’ developed by the European project Venus-C is used to perform various monitoring tasks and house-keeping duties, such as moving input model data from the storage service to the compute service (see <http://www.venus-c.eu>).

A key feature of Cloud Computing is that applications are web-based and many of the activities described here are hidden from the end-user. Figure 12.58 shows a page from the web site that allows users to modify the random field parameters, launch jobs, monitor their progress and view aggregated results.

One of the criticisms of Cloud Computing is that computations are often carried out on ‘virtual machines’, software implementations of real computers. The use of virtual machines ensures the portability of Cloud applications across all the different types of real hardware found in a typical data centre. However, this comes at a significant performance penalty as shown in Figure 12.59. A single analysis or ‘realisation’ takes 10 times longer on Azure than on a single core of a Cray XE6.



**Figure 12.58** Cloud application job submission web page

# Elements	Azure (s)	Cray XE6 (s)
82,944	65	6
196,608	160	18
384,000	374	38
663,552	722	78
1,572,864	8,906	798
5,308,416	11,985	1,558

**Figure 12.59** Performance: Microsoft Azure Cloud versus Cray XE6

Virtual Machine			Cost (Month)	Cost (Hour)
Size	Cores	Memory (GB)		
Extra Small	Shared	0.76	\$9.36	\$0.013
Small	1	1.75	\$57.60	\$0.08
Medium	2	3.5	\$115.20	\$0.16
Large	4	7	\$230.40	\$0.32
Extra Large	8	14	\$460.80	\$0.64

**Figure 12.60** Typical costs: Microsoft Azure

That said, the service being offered by Microsoft Azure is based on ‘elasticity in demand’ rather than absolute performance. Tens or hundreds of thousands of virtual machines can be requested for very short durations of time, thus providing short engineering design turnaround times without the need to invest in a supercomputer. Figure 12.60 gives typical costs for Azure, c. 2012, showing that ‘sensitivity analysis’ or ‘Monte Carlo’ simulations are very affordable.

## 12.5 Conclusions

All the serial program types described in Chapters 5–11 have been parallelised using a consistent strategy. Performance data are excellent for a certain genre of current ‘supercomputer’, c. 2012. Analyses have ‘scaled’ well for up to a billion equations and 16,000 processes.

Current research in supercomputing is aimed at building an Exascale computer, one that is capable of performing  $10^{18}$  operations per second, by the year 2020. It is thought that Exascale computations will use many millions of threads of execution. In that case, we can anticipate that the programs provided herein will eventually require a new parallelisation strategy. One option could be mixed OpenMP/MPI aimed at future ‘many integrated core’ processors. MPI could facilitate communication between tens of thousands of processors, whilst OpenMP could operate on the hundreds of cores that are expected to be available on each processor. GPUs could also be used as another tier of execution, as accelerators for numerically intensive computations, but it is possible that GPU hardware will be integrated with the host processor in the future.

In terms of engineering analysis, we have shown that the processing capability of supercomputers now allows engineers to study structures with complex architectures, with the geometry being captured by 3D imaging. Cloud Computing brings sensitivity analysis within reach of most engineering firms, at a reasonable cost.

## 12.6 Glossary of Variable Names

### Scalar integers:

cjters	conjugate gradient iteration counter
cjits	conjugate gradient iteration ceiling
cjtots	total number of conjugate gradient iterations
cj_tot	total number of BiCGStab iterations
ell	1 in the BiCGStab(l) process
fixed_freedoms	number of fixed freedoms
i	simple counter
iel	element counter
iflag	failure flag
incs	number of load increments
is	location of desired output freedom
it	processor on which desired output resides
iters	iteration counter
iy	simple counter
j	simple counter
jflag	failure flag
jj	simple counter
k	simple counter
l	simple counter
lalfa	length of alfa array
leig	length of eig array
limit	iteration ceiling
loaded_freedoms	number of loaded freedoms
loaded_nodes	number of loaded nodes
lp	output channel number
lx	problem-dependent array size
lz	problem-dependent array size
ndim	number of dimensions
neig	problem-dependent array size
neq_temp	temporary equation sum

nip	number of integrating points
nle	number of loaded elements (side of square)
nmodes	number of eigenmodes computed
nn_temp	temporary node sum
nod	number of nodes per element
nodf	number of pressure degrees of freedom per element
nodof	number of degrees of freedom per node
no_index_start	start address of loaded/fixed freedoms
npri	print interval
nr	number of restrained nodes
nres	number of output freedom
nst	number of stress/strain terms
nstep	number of time steps in analysis
num_no	number of processors holding load/displacement data
nxe	number of elements in $x$ -direction
nye	number of elements in $y$ -direction
nze	number of elements in $z$ -direction
n_t	total number of degrees of freedom per element
plasiters	plastic iteration counter
plasits	plastic iteration ceiling

**Scalar reals:**

aa	$x$ -dimension of elements
acc	accuracy parameter
alpha	pcg parameter
alpha1	Rayleigh damping parameter
bb	$y$ -dimension of elements
beta	pcg parameter
beta1	Rayleigh damping parameter
big	largest component of a vector
c	cohesion
cc	$z$ -dimension of elements
cjtol	conjugate gradient iteration tolerance
c1	
c2	intermediate reals
c3	
c4	
det	determinant of Jacobian matrix
dq1	Mohr–Coulomb plastic potential derivative
dq2	Mohr–Coulomb plastic potential derivative
dq3	Mohr–Coulomb plastic potential derivative
dsbar	shear stress invariant
dt	viscoplastic ‘time’ step
dtim	time step
e	Young’s modulus
el	left limit of eigenvalue spectrum
er	right limit of eigenvalue spectrum

error	residual error
f	current stress state
fac	yield factor
fnew	new yield function
gama	intermediate value
kappa	kappa in BiCGStab process
kx	conductivity in $x$ -direction
ky	conductivity in $y$ -direction
kz	conductivity in $z$ -direction
lode_theta	Lode angle
norm_r	norm of residual
omega	intermediate value
period	period of oscillation
phi	angle of internal friction
plastol	plastic iteration tolerance
pload	load multiple
pp	intermediate value
presc	prescribed value of load/displacement
psi	angle of dilation
q	total load
rho	density
rho1	intermediate value
r0_norm	starting residual norm
sigm	mean stress invariant
sinph	sine of angle of internal friction
penalty	value of penalty restraint
period	period of oscillation
phi	angle of internal friction
pload	load multiple
real_time	accumulated time
r0_norm	starting residual norm
sbar	shear stress
sbary	shear yield stress
theta	parameter in ‘theta’ integrator
tol	convergence tolerance
ubar	average $x$ -velocity
up	pcg parameter
v	Poisson’s ratio
val0	initial value
vbar	average $y$ -velocity
visc	viscosity
volume	element volume
wbar	average $z$ -velocity
x0	start value

**Scalar characters:**

element	element type
---------	--------------

**Scalar logicals:**

cj_converged	set to .TRUE. if conjugate gradient iterations converged
consistent	set to .TRUE. if element mass is ‘consistent’
converged	set to .TRUE. if solution
plastic_converged	set to .TRUE. if plastic iterations converged

**Dynamic integer arrays:**

b_pp	distributed right-hand-side vector
g	element steering vector
g_g_pp	distributed global steering vector
g_num_pp	distributed global node numbers
g_t	total element $g$ -vector
jeig	intermediate array
no	freedoms to be loaded/fixed
no_f	vector of fixed freedom numbers
no_local	local (processor) freedoms
no_local_temp	temporary store
nu	intermediate array
num	element node numbers
rest	node freedom restraints

**Dynamic real arrays:**

alfa	intermediate array
ans_pp	distributed answer vector
bdylds_pp	distributed body-loads vector
bee	strain-displacement matrix
beta	intermediate array
bload	element body loads
c	coupling matrix
col	column array
coord	element nodal coordinates
coordf	nodal coordinates of pressure nodes
c11	
c12	
c21	
c23	c arrays—see equation (2.115)
c32	
c24	
c42	
dee	stress-strain matrix
del	intermediate array
der	derivatives wrt local coordinates
derf	local derivatives of pressure shape functions
deriv	derivatives wrt global coordinates
derivf	global derivatives of pressure shape functions
devp	increment of viscoplastic strain
diag_pp	distributed diagonal vector
diag_precon_pp	distributed diagonal preconditioning matrix

diag_precon_tmp	temporary store
d1x0_pp	distributed old velocity vector
d1x1_pp	distributed new velocity vector
d2x0_pp	distributed old acceleration vector
d2x1_pp	distributed new acceleration vector
d_pp	distributed pcg vector
ecm	element consistent mass matrix
eig	intermediate array
eld	element nodal displacements
eld_pp	distributed nodal displacements
eload	accumulating element body loads
eps	element strains
erate	viscoplastic strain rate
etensor_pp	distributed element strains
evp	viscoplastic strains
evpt_pp	distributed total viscoplastic strains
flow	viscoplastic flow
fun	element shape functions
funf	intermediate array
funny	intermediate array
funnyf	intermediate array
gamma	intermediate array
gg	intermediate array
globma_pp	distributed global mass matrix
globma_tmp	temporary storage mass vector
jac	Jacobian matrix
kay	conductivity matrix
kc	conductivity matrix
kcx	$x$ contribution to conductivity matrix
kcy	$y$ contribution to conductivity matrix
kcz	$z$ contribution to conductivity matrix
kd	total element matrix
ke	element ‘stiffness’ matrix
km	element stiffness matrix
loads_pp	distributed loads vector
mm_pp	distributed mass vector
mm_tmp	temporary vector
m1	plastic potential derivatives matrix
m2	plastic potential derivatives matrix
m3	plastic potential derivatives matrix
new_lo_pp	distributed new loads
oldis_pp	previous distributed displacements
pm	element mass matrix
points	integrating point local coordinates
p_g_co_pp	distributed nodal coordinates
p_mul_pp	gather-scatter matrix
p_pp	distributed pcg vector
qinc	vector of load increment terms

row	row array
row1	
row2	intermediate arrays
row3	
rt_pp	distributed vector
r_pp	distributed pcg vector
s	intermediate array
sigma	element stresses
store_pp	distributed penalty storage
storka_pp	distributed storage of pm and km
storkb_pp	distributed storage of pm and km
storkm_pp	distributed stored element stiffness matrices
stress	stress vector
temp_pp	distributed intermediate vector
tensor_pp	distributed stresses
totd_pp	distributed total displacements
ua_pp	distributed $\{U\}$ in product $\{U\} + [A]\{V\}$
utemp_pp	gather-scatter matrix
uvel	$x$ -velocity
u_pp	distributed pcg vector
val	prescribed load/displacement values
vvel	$y$ -velocity
val_f	values of fixed freedoms
va_pp+	distributed $\{V\}$ in product $\{U\} + [A]\{V\}$
vdiag_pp	distributed
vol	array for volumetric strain
volf	array for fluid volumetric strain
vu_pp	distributed intermediate vector
v_store_pp	distributed stored Lanczos vectors
weights	weighting coefficients
wvel	$z$ -velocity
w1_pp	intermediate array
xnew_pp	distributed pcg vector
x1_pp	distributed new displacement vector
x_old_pp	distributed previous $x$ -vector
x_pp	distributed pcg vector
y_pp	distributed $y$ -vector
y1_pp	distributed $y_1$ -vector
z_pp	intermediate array

## References

- Brassey CA, Margetts L, Kitchener AC, Withers PJ, Manning PL and Sellers WI 2013 Finite element modelling versus classic beam theory: Comparing methods for stress estimation in a morphologically diverse sample of vertebrate long bones. *R Soc Interface*, **10**(79).
- Chan KH, Ligang L and Xinhao L 2006 Modelling the core convection using finite element and finite difference methods. *Phys Earth Planet Interior* **157**(1/2), 124–138.
- Chan KH, Zhang K, Li L and Liao X 2007 A new generation of convection–driven spherical dynamos using EBE finite element methods. *Phys Earth Planet Interior* **163**(1–4), 251–265.

- Christias D 2012 *vProduct: Virtual prototyping on-demand using cloud computing technology*. MSc Thesis, University of Manchester.
- Evans LL, Margetts L, Dudarev S, Young P and Mummary PM 2013 Parallel processing for time-dependent heat flow problems. *Proc NAFEMS World Congress*, NAFEMS, Austria.
- Falkingham PF, Bates KT, Margetts L and Manning PL 2011 The ‘Goldilocks’ effect: Preservational bias in vertebrate track assemblages. *R Soc Interface* **8**(61), 1142–1154.
- Falkingham PF, Margetts L, Smith IM and Manning PL 2009 A reinterpretation of palmate and semi-palmate (webbed) fossil tracks: Insights from finite element modelling. *Palaeogeogr Palaeoclim Palaeoecol* **271**(1/2), 69–76.
- Fenton GA and Griffiths DV 2008 *Risk Assessment in Geotechnical Engineering*. John Wiley & Sons, Chichester.
- Karypis G and Kumar V 1999 Parallel multilevel k-way partitioning scheme for irregular graphs. *Siam Rev* **41**(2), 278–300.
- Margetts L 2002 *Parallel finite element analysis*. PhD thesis, University of Manchester, UK.
- 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. In *High-Performance Computing and Networking: Lecture Notes in Computer Science*. Springer-Verlag, Berlin, pp. 400–409.
- Smith IM and Margetts L 2003 Portable parallel processing for nonlinear problems. *Proc COMPLAS 2003*, CIMNE, Barcelona.
- Smith IM and Margetts L 2006 The convergence variability of parallel iterative solvers. *Eng Comput* **23**(2), 154–165.
- Smith IM and Wang A 1998 Analysis of piled rafts. *Int J Numer Anal Methods Geomech* **22**(10), 777–790.

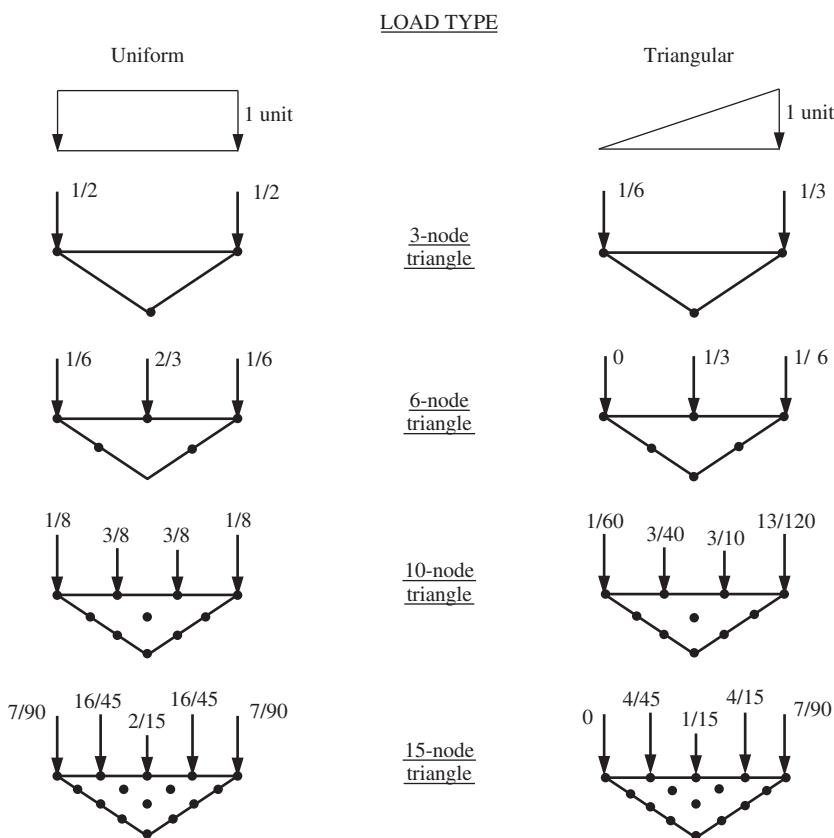


# Appendix A

## Equivalent Nodal Loads

### Planar Elements (2D)

Width of loaded face = 1 unit



## Planar Elements (2D)

Width of loaded face = 1 unit

### LOAD TYPE

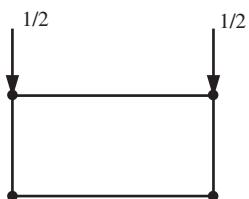
Uniform



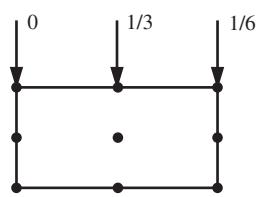
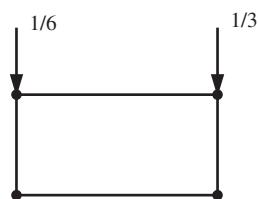
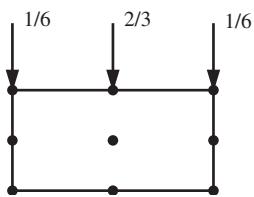
Triangular



4-node quadrilateral

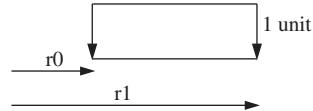
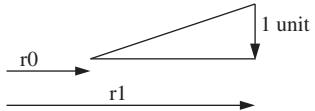
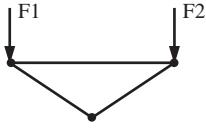
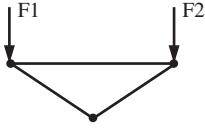
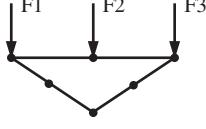
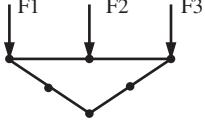


8- and 9-node quadrilaterals



## Axisymmetric Elements (2D)

Loading over 1 radian

Uniform	LOAD TYPE	Triangular
	<b>LOAD TYPE</b> <b>Uniform</b>	
	<u>3-node triangle</u>	
$F_1 = \frac{1}{6} (r_1^2 + r_0 r_1 - 2r_0^2)$ $F_2 = \frac{1}{6} (2r_1^2 - r_0 r_1 - r_0^2)$		$F_1 = \frac{1}{12} (r_1^2 - r_0^2)$ $F_2 = \frac{1}{12} (3r_1^2 - 2r_0 r_1 - r_0^2)$
	<u>6-node triangle</u>	
$F_1 = \frac{1}{6} (r_0 r_1 - r_0^2)$ $F_2 = \frac{1}{3} (r_1^2 - r_0^2)$ $F_3 = \frac{1}{6} (r_1^2 - r_0 r_1)$		$F_1 = -\frac{1}{60} (r_1^2 - 2r_0 r_1 + r_0^2)$ $F_2 = \frac{1}{15} (3r_1^2 - r_0 r_1 - 2r_0^2)$ $F_3 = \frac{1}{60} (9r_1^2 - 8r_0 r_1 - r_0^2)$

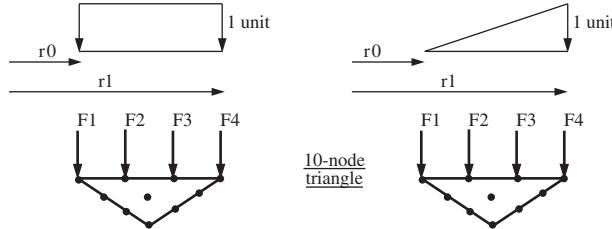
## Axisymmetric Elements (2D)

Loading over 1 radian

### LOAD TYPE

Uniform

Triangular

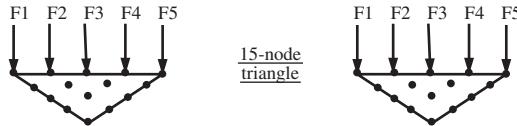


$$F_1 = \frac{1}{120} \left( 2r_1^2 + 11r_0r_1 - 13r_0^2 \right) \quad F_1 = \frac{1}{120} \left( r_1^2 - r_0^2 \right)$$

$$F_2 = \frac{1}{40} \left( 3r_1^2 + 9r_0r_1 - 12r_0^2 \right) \quad F_2 = \frac{3}{40} \left( r_0r_1 - r_0^2 \right)$$

$$F_3 = \frac{1}{40} \left( 12r_1^2 - 9r_0r_1 - 3r_0^2 \right) \quad F_3 = \frac{3}{40} \left( 3r_1^2 - 2r_0r_1 - r_0^2 \right)$$

$$F_4 = \frac{1}{120} \left( 13r_1^2 - 11r_0r_1 - 2r_0^2 \right) \quad F_4 = \frac{1}{120} \left( 12r_1^2 - 11r_0r_1 - r_0^2 \right)$$



$$F_1 = \frac{7}{90} \left( r_0r_1 - r_0^2 \right) \quad F_1 = -\frac{1}{252} \left( r_1^2 - 2r_0r_1 + r_0^2 \right)$$

$$F_2 = \frac{4}{45} \left( r_1^2 + 2r_0r_1 - 3r_0^2 \right) \quad F_2 = \frac{4}{315} \left( 3r_1^2 + r_0r_1 - 4r_0^2 \right)$$

$$F_3 = \frac{1}{15} \left( r_1^2 - r_0^2 \right) \quad F_3 = \frac{1}{105} \left( r_1^2 + 5r_0r_1 - 6r_0^2 \right)$$

$$F_4 = \frac{4}{45} \left( 3r_1^2 - 2r_0r_1 - r_0^2 \right) \quad F_4 = \frac{1}{315} \left( 17r_1^2 - 13r_0r_1 - 4r_0^2 \right)$$

$$F_5 = \frac{7}{90} \left( r_1^2 - r_0r_1 \right) \quad F_5 = \frac{1}{1260} \left( 93r_1^2 - 88r_0r_1 - 5r_0^2 \right)$$

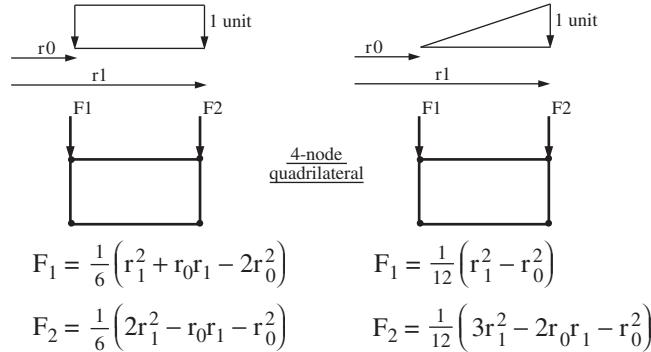
## Axisymmetric Elements (2D)

Loading over 1 radian

### LOAD TYPE

Uniform

Triangular



The diagram shows a quadrilateral element with three nodes on the bottom edge labeled F1, F2, and F3. The central node is marked with a dot.

**8 and 9-node quadrilaterals:**

$$F_1 = \frac{1}{6} (r_0 r_1 - r_0^2)$$

$$F_2 = \frac{1}{3} (r_1^2 - r_0^2)$$

$$F_3 = \frac{1}{6} (r_1^2 - r_0 r_1)$$

$$F_1 = -\frac{1}{60} (r_1^2 - 2r_0 r_1 + r_0^2)$$

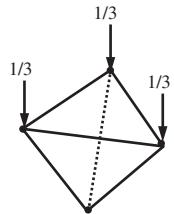
$$F_2 = \frac{1}{15} (3r_1^2 - r_0 r_1 - 2r_0^2)$$

$$F_3 = \frac{1}{60} (9r_1^2 - 8r_0 r_1 - r_0^2)$$

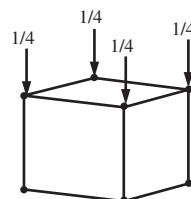
### Three Dimensional Elements (3D)

Area of loaded face = 1 unit  
Unit stress applied

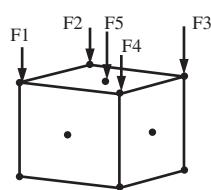
4-node  
tetrahedron



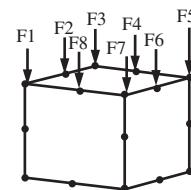
8-node  
hexahedron



14-node  
hexahedron (type 5)



20-node  
hexahedron



$$F_1 = F_2 = F_3 = F_4 = \frac{1}{12}$$

$$F_5 = \frac{2}{3}$$

$$F_1 = F_3 = F_5 = F_7 = -\frac{1}{12}$$

$$F_2 = F_4 = F_6 = F_8 = \frac{1}{3}$$

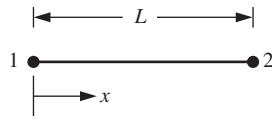
# Appendix B

## Shape Functions and Element Node Numbering

### 1D Elements

#### 2-node rod

$$N_1 = 1 - \frac{x}{L}$$
$$N_2 = \frac{x}{L}$$



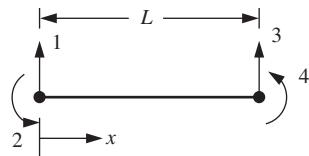
#### 2-node beam

$$N_1 = \frac{1}{L^3}(L^3 - 3Lx^2 + 2x^3)$$

$$N_2 = \frac{1}{L^2}(L^2x - 2Lx^2 + x^3)$$

$$N_3 = \frac{1}{L^3}(3Lx^2 - 2x^3)$$

$$N_4 = \frac{1}{L^2}(x^3 - Lx^2)$$



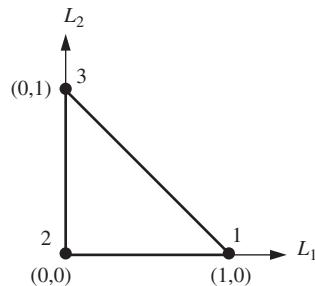
## 2D Elements

### 3-node triangle

$$N_1 = L_1$$

$$N_2 = (1 - L_1 - L_2)$$

$$N_3 = L_2$$



### 6-node triangle

$$N_1 = (2L_1 - 1)L_1$$

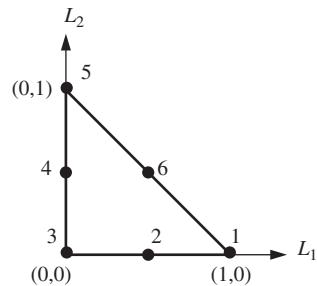
$$N_2 = 4(1 - L_1 - L_2)L_1$$

$$N_3 = (2(1 - L_1 - L_2) - 1)(1 - L_1 - L_2)$$

$$N_4 = 4L_2(1 - L_1 - L_2)$$

$$N_5 = (2L_2 - 1)L_2$$

$$N_6 = 4L_1L_2$$



### 10-node triangle

$$N_1 = \frac{1}{2}(3L_1 - 1)(3L_1 - 2)L_1$$

$$N_2 = -\frac{9}{2}(L_2 + L_1 - 1)(3L_1 - 1)L_1$$

$$N_3 = \frac{9}{2}(3L_2 + 3L_1 - 2)(L_2 + L_1 - 1)L_1$$

$$N_4 = -\frac{1}{2}(3L_2 + 3L_1 - 1)(3L_2 + 3L_1 - 2)(L_2 + L_1 - 1)$$

$$N_5 = \frac{9}{2}(3L_2 + 3L_1 - 2)(L_2 + L_1 - 1)L_2$$

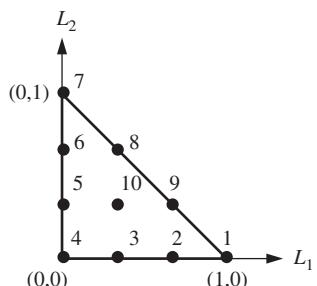
$$N_6 = -\frac{9}{2}(3L_2 - 1)(L_2 + L_1 - 1)L_2$$

$$N_7 = \frac{1}{2}(3L_2 - 1)(3L_2 - 2)L_2$$

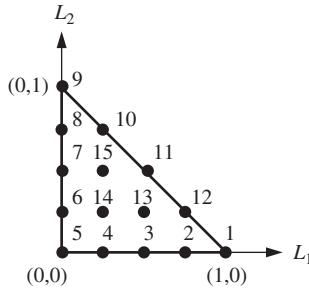
$$N_8 = \frac{9}{2}(3L_2 - 1)L_2L_1$$

$$N_9 = \frac{9}{2}(3L_1 - 1)L_2L_1$$

$$N_{10} = -27(L_2 + L_1 - 1)L_2L_1$$



### 15-node triangle



$$N_1 = \frac{32}{3} L_1 (L_1 - \frac{1}{4})(L_1 - \frac{1}{2})(L_1 - \frac{3}{4})$$

$$N_2 = \frac{128}{3} L_1 (1 - L_1 - L_2)(L_1 - \frac{1}{4})(L_1 - \frac{1}{2})$$

$$N_3 = 64 L_1 (1 - L_1 - L_2)(L_1 - \frac{1}{4})(1 - L_2 - L_1 - \frac{1}{4})$$

$$N_4 = \frac{128}{3} L_1 (1 - L_1 - L_2)(1 - L_2 - L_1 - \frac{1}{4})(1 - L_2 - L_1 - \frac{1}{2})$$

$$N_5 = \frac{32}{3} (1 - L_1 - L_2)(1 - L_2 - L_1 - \frac{1}{4})(1 - L_2 - L_1 - \frac{1}{2})(1 - L_2 - L_1 - \frac{3}{4})$$

$$N_6 = \frac{128}{3} (1 - L_1 - L_2)L_2 (1 - L_2 - L_1 - \frac{1}{4})(1 - L_2 - L_1 - \frac{1}{2})$$

$$N_7 = 64 (1 - L_1 - L_2)L_2 (L_2 - \frac{1}{4})(1 - L_1 - L_2 - \frac{1}{4})$$

$$N_8 = \frac{128}{3} (1 - L_1 - L_2)L_2 (L_2 - \frac{1}{4})(L_2 - \frac{1}{2})$$

$$N_9 = \frac{32}{3} L_2 (L_2 - \frac{1}{4})(L_2 - \frac{1}{2})(L_2 - \frac{3}{4})$$

$$N_{10} = \frac{128}{3} L_2 L_1 (L_2 - \frac{1}{4})(L_2 - \frac{1}{2})$$

$$N_{11} = 64 L_2 L_1 (L_2 - \frac{1}{4})(L_1 - \frac{1}{4})$$

$$N_{12} = \frac{128}{3} L_2 L_1 (L_1 - \frac{1}{4})(L_1 - \frac{1}{2})$$

$$N_{13} = 128 L_2 L_1 (1 - L_1 - L_2)(L_1 - \frac{1}{4})$$

$$N_{14} = 128 L_2 L_1 (1 - L_1 - L_2)(1 - L_2 - L_1 - \frac{1}{4})$$

$$N_{15} = 128 L_2 L_1 (L_2 - \frac{1}{4})(1 - L_1 - L_2)$$

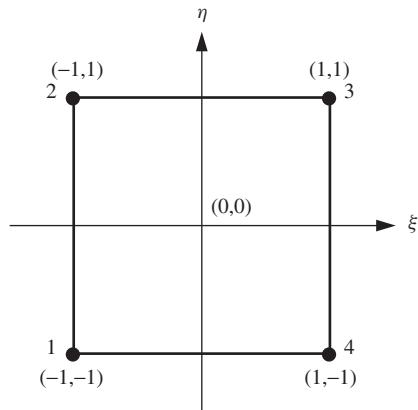
**4-node quadrilateral**

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

$$N_2 = \frac{1}{4}(1 - \xi)(1 + \eta)$$

$$N_3 = \frac{1}{4}(1 + \xi)(1 + \eta)$$

$$N_4 = \frac{1}{4}(1 + \xi)(1 - \eta)$$

**8-node quadrilateral**

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

$$N_2 = \frac{1}{2}(1 - \xi)(1 - \eta^2)$$

$$N_3 = \frac{1}{4}(1 - \xi)(1 + \eta)(-\xi + \eta - 1)$$

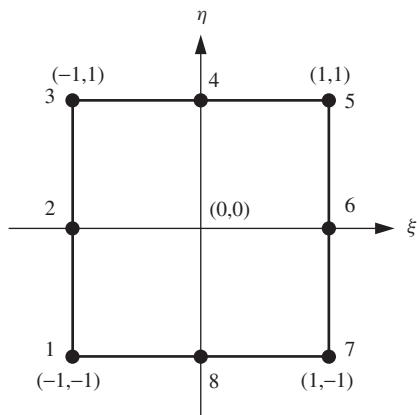
$$N_4 = \frac{1}{2}(1 - \xi^2)(1 + \eta)$$

$$N_5 = \frac{1}{4}(1 + \xi)(1 + \eta)(\xi + \eta - 1)$$

$$N_6 = \frac{1}{2}(1 + \xi)(1 - \eta^2)$$

$$N_7 = \frac{1}{4}(1 + \xi)(1 - \eta)(\xi - \eta - 1)$$

$$N_8 = \frac{1}{2}(1 - \xi^2)(1 - \eta)$$

**9-node quadrilateral**

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

$$N_2 = -\frac{1}{2}\xi(\xi - 1)(\eta + 1)(\eta - 1)$$

$$N_3 = \frac{1}{4}\xi(\xi - 1)\eta(\eta + 1)$$

$$N_4 = -\frac{1}{2}(\xi + 1)(\xi - 1)\eta(\eta + 1)$$

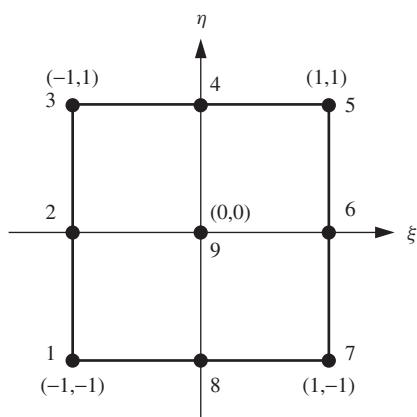
$$N_5 = \frac{1}{4}\xi(\xi + 1)\eta(\eta + 1)$$

$$N_6 = -\frac{1}{2}\xi(\xi + 1)(\eta + 1)(\eta - 1)$$

$$N_7 = \frac{1}{4}\xi(\xi + 1)\eta(\eta - 1)$$

$$N_8 = -\frac{1}{2}(\xi + 1)(\xi - 1)\eta(\eta - 1)$$

$$N_9 = (\xi + 1)(\xi - 1)(\eta + 1)(\eta - 1)$$



### 3D Elements

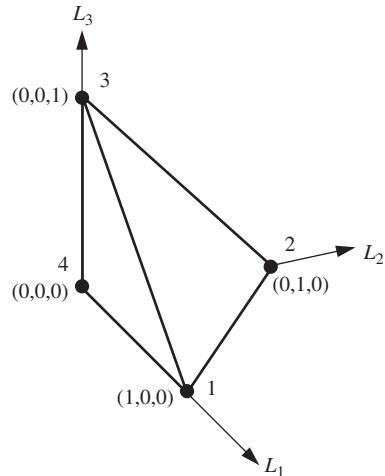
#### 4-node tetrahedron

$$N_1 = L_1$$

$$N_2 = L_2$$

$$N_3 = L_3$$

$$N_4 = (1 - L_1 - L_2 - L_3)$$



#### 8-node hexahedron

$$N_1 = \frac{1}{8}(1 - \xi)(1 - \eta)(1 - \zeta)$$

$$N_2 = \frac{1}{8}(1 - \xi)(1 - \eta)(1 + \zeta)$$

$$N_3 = \frac{1}{8}(1 + \xi)(1 - \eta)(1 + \zeta)$$

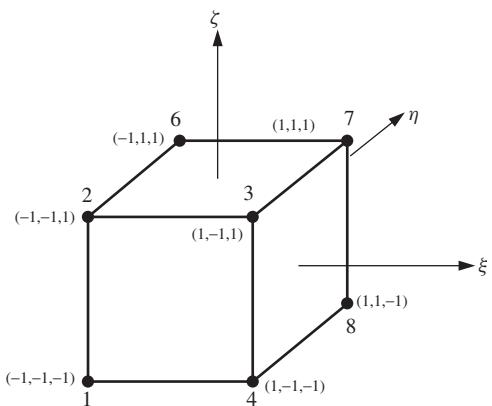
$$N_4 = \frac{1}{8}(1 + \xi)(1 - \eta)(1 - \zeta)$$

$$N_5 = \frac{1}{8}(1 - \xi)(1 + \eta)(1 - \zeta)$$

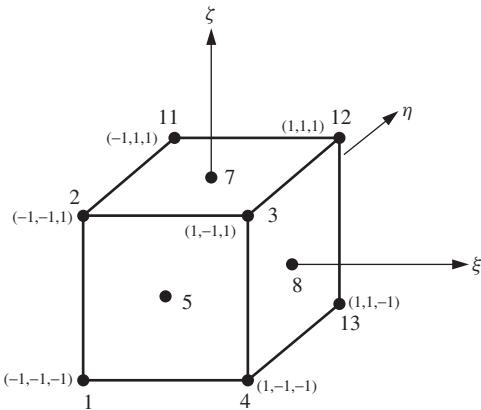
$$N_6 = \frac{1}{8}(1 - \xi)(1 + \eta)(1 + \zeta)$$

$$N_7 = \frac{1}{8}(1 + \xi)(1 + \eta)(1 + \zeta)$$

$$N_8 = \frac{1}{8}(1 + \xi)(1 + \eta)(1 - \zeta)$$



### 14-node hexahedron (Type 6)



$$N_1 = \frac{1}{8}(\xi\eta + \xi\zeta + 2\xi + \eta\zeta + 2\eta + 2\zeta + 2)(\xi - 1)(\eta - 1)(\zeta - 1)$$

$$N_2 = -\frac{1}{8}(\xi\eta - \xi\zeta + 2\xi - \eta\zeta + 2\eta - 2\zeta + 2)(\xi - 1)(\eta - 1)(\zeta + 1)$$

$$N_3 = -\frac{1}{8}(\xi\eta - \xi\zeta + 2\xi + \eta\zeta - 2\eta + 2\zeta - 2)(\xi + 1)(\eta - 1)(\zeta + 1)$$

$$N_4 = \frac{1}{8}(\xi\eta + \xi\zeta + 2\xi - \eta\zeta - 2\eta - 2\zeta - 2)(\xi + 1)(\eta - 1)(\zeta - 1)$$

$$N_5 = -\frac{1}{2}(\xi + 1)(\xi - 1)(\eta - 1)(\zeta + 1)(\zeta - 1)$$

$$N_6 = -\frac{1}{2}(\xi - 1)(\eta + 1)(\eta - 1)(\zeta + 1)(\zeta - 1)$$

$$N_7 = \frac{1}{2}(\xi + 1)(\xi - 1)(\eta + 1)(\eta - 1)(\zeta + 1)$$

$$N_8 = \frac{1}{2}(\xi + 1)(\eta + 1)(\eta - 1)(\zeta + 1)(\zeta - 1)$$

$$N_9 = -\frac{1}{2}(\xi + 1)(\xi - 1)(\eta + 1)(\eta - 1)(\zeta - 1)$$

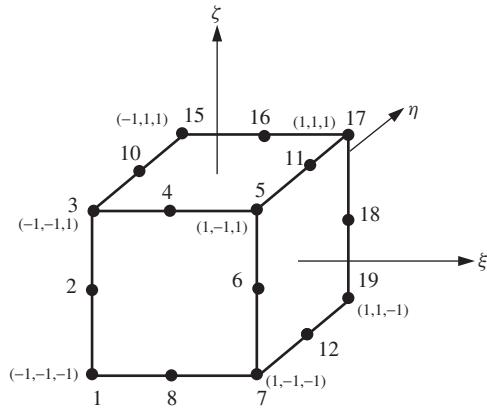
$$N_{10} = \frac{1}{8}(\xi\eta - \xi\zeta - 2\xi + \eta\zeta + 2\eta - 2\zeta - 2)(\xi - 1)(\eta + 1)(\zeta - 1)$$

$$N_{11} = -\frac{1}{8}(\xi\eta + \xi\zeta - 2\xi - \eta\zeta + 2\eta + 2\zeta - 2)(\xi - 1)(\eta + 1)(\zeta + 1)$$

$$N_{12} = -\frac{1}{8}(\xi\eta + \xi\zeta - 2\xi + \eta\zeta - 2\eta - 2\zeta + 2)(\xi + 1)(\eta + 1)(\zeta + 1)$$

$$N_{13} = \frac{1}{8}(\xi\eta - \xi\zeta - 2\xi - \eta\zeta - 2\eta + 2\zeta + 2)(\xi + 1)(\eta + 1)(\zeta - 1)$$

$$N_{14} = \frac{1}{2}(\xi + 1)(\xi - 1)(\eta + 1)(\zeta + 1)(\zeta - 1)$$

**20-node hexahedron**

$$N_1 = \frac{1}{8}(1 - \xi)(1 - \eta)(1 - \zeta)(-\xi - \eta - \zeta - 2)$$

$$N_2 = \frac{1}{4}(1 - \xi)(1 - \eta)(1 - \zeta^2)$$

$$N_3 = \frac{1}{8}(1 - \xi)(1 - \eta)(1 + \zeta)(-\xi - \eta + \zeta - 2)$$

$$N_4 = \frac{1}{4}(1 - \xi^2)(1 - \eta)(1 + \zeta)$$

$$N_5 = \frac{1}{8}(1 + \xi)(1 - \eta)(1 + \zeta)(\xi - \eta + \zeta - 2)$$

$$N_6 = \frac{1}{4}(1 + \xi)(1 - \eta)(1 - \zeta^2)$$

$$N_7 = \frac{1}{8}(1 + \xi)(1 - \eta)(1 - \zeta)(\xi - \eta - \zeta - 2)$$

$$N_8 = \frac{1}{4}(1 - \xi^2)(1 - \eta)(1 - \zeta)$$

$$N_9 = \frac{1}{4}(1 - \xi)(1 - \eta^2)(1 - \zeta)$$

$$N_{10} = \frac{1}{4}(1 - \xi)(1 - \eta^2)(1 + \zeta)$$

$$N_{11} = \frac{1}{4}(1 + \xi)(1 - \eta^2)(1 + \zeta)$$

$$N_{12} = \frac{1}{4}(1 + \xi)(1 - \eta^2)(1 - \zeta)$$

$$N_{13} = \frac{1}{8}(1 - \xi)(1 + \eta)(1 - \zeta)(-\xi + \eta - \zeta - 2)$$

$$N_{14} = \frac{1}{4}(1 - \xi)(1 + \eta)(1 - \zeta^2)$$

$$N_{15} = \frac{1}{8}(1 - \xi)(1 + \eta)(1 + \zeta)(-\xi + \eta + \zeta - 2)$$

$$N_{16} = \frac{1}{4}(1 - \xi^2)(1 + \eta)(1 + \zeta)$$

$$N_{17} = \frac{1}{8}(1 + \xi)(1 + \eta)(1 + \zeta)(\xi + \eta + \zeta - 2)$$

$$N_{18} = \frac{1}{4}(1 + \xi)(1 + \eta)(1 - \zeta^2)$$

$$N_{19} = \frac{1}{8}(1 + \xi)(1 + \eta)(1 - \zeta)(\xi + \eta - \zeta - 2)$$

$$N_{20} = \frac{1}{4}(1 - \xi^2)(1 + \eta)(1 - \zeta)$$



# Appendix C

## Plastic Stress-Strain Matrices and Plastic Potential Derivatives

The following expressions give the plastic stress-strain matrices for 2D applications using von Mises (Yamada *et al.*, 1968) and Mohr–Coulomb (Griffiths and Willson, 1986; see Chapter 6 references). For 3D applications, the expressions are more conveniently generated using computer algebra systems (see subroutines `vmdpl` and `mcdpl` for von Mises and Mohr–Coulomb, respectively).

### A. Plastic stress–strain matrices

#### 1. Von Mises

$$[\mathbf{D}^p] = \frac{2G}{t^2} \begin{bmatrix} s_x^2 & s_x s_y & s_x \tau_{xy} & s_z s_x \\ s_y^2 & s_y \tau_{xy} & s_y s_z & \\ \tau_{xy}^2 & s_z \tau_{xy} & s_z^2 \\ \text{symmetrical} & s_z^2 & \end{bmatrix}$$

where

$G$  = shear modulus

$t$  = second deviatoric stress invariant (6.3)

$s_x = (2\sigma_x - \sigma_y - \sigma_z)/3$ , etc.

#### 1. Mohr–Coulomb

If not near a corner, that is  $|\sin \theta| \leq 0.49$ , then

$$[\mathbf{D}^p] = \frac{E}{2(1+\nu)(1-2\nu)(1-2\nu + \sin \phi \sin \psi)} [\mathbf{A}]$$

where

$$[\mathbf{A}] = \begin{bmatrix} R_1 C_1 & R_1 C_2 & R_1 C_3 & R_1 C_4 \\ R_2 C_1 & R_2 C_2 & R_2 C_3 & R_2 C_4 \\ R_3 C_1 & R_3 C_2 & R_3 C_3 & R_3 C_4 \\ R_4 C_1 & R_4 C_2 & R_4 C_3 & R_4 C_4 \end{bmatrix},$$

$$\begin{aligned}
C_1 &= \sin \phi + k_1(1 - 2\nu) \sin \alpha \\
C_2 &= \sin \phi - k_1(1 - 2\nu) \sin \alpha \\
C_3 &= k_2(1 - 2\nu) \cos \alpha \\
C_4 &= 2\nu \sin \phi \\
R_1 &= \sin \psi + k_1(1 - 2\nu) \sin \alpha \\
R_2 &= \sin \psi - k_1(1 - 2\nu) \sin \alpha \\
R_3 &= k_2(1 - 2\nu) \cos \alpha \\
R_4 &= 2\nu \sin \psi
\end{aligned} \tag{C.1}$$

$$\alpha = \arctan \left| \frac{\sigma_x - \sigma_y}{2\tau_{xy}} \right|$$

and

$$\begin{aligned}
k_1 &= \begin{cases} 1 & \text{if } |\sigma_y| \geq |\sigma_x| \\ -1 & \text{if } |\sigma_y| < |\sigma_x| \end{cases} \\
k_2 &= \begin{cases} 1 & \text{if } \tau_{xy} \geq 0 \\ -1 & \text{if } \tau_{xy} < 0 \end{cases}
\end{aligned}$$

If near a corner, that is  $|\sin \theta| > 0.49$ , then

$$[\mathbf{D}^p] = \frac{E}{(1 + \nu)(1 - 2\nu)(K_\phi \sin \psi + C_\phi C_\psi t^2(1 - 2\nu))} [\mathbf{A}]$$

where  $[\mathbf{A}]$  is defined as before with

$$\begin{aligned}
C_1 &= K_\phi + C_\phi((1 - \nu)s_x + \nu(s_y + s_z)) \\
C_2 &= K_\phi + C_\phi((1 - \nu)s_y + \nu(s_z + s_x)) \\
C_3 &= C_\phi(1 - 2\nu)\tau_{xy} \\
C_4 &= K_\phi + C_\phi((1 - \nu)s_z + \nu(s_x + s_y)) \\
R_1 &= K_\psi + C_\psi((1 - \nu)s_x + \nu(s_y + s_z)) \\
R_2 &= K_\psi + C_\psi((1 - \nu)s_y + \nu(s_z + s_x)) \\
R_3 &= C_\psi(1 - 2\nu)\tau_{xy} \\
R_4 &= K_\psi + C_\psi((1 - \nu)s_z + \nu(s_x + s_y))
\end{aligned} \tag{C.2}$$

and

$$K_\phi = \frac{\sin \phi}{3}(1 + \nu)$$

$$K_\psi = \frac{\sin \psi}{3}(1 + \nu)$$

$$C_\phi = \frac{\sqrt{6}}{4t} \left( 1 \pm \frac{\sin \phi}{3} \right)$$

$$C_\psi = \frac{\sqrt{6}}{4t} \left( 1 \pm \frac{\sin \psi}{3} \right)$$

In the expressions for  $C_\phi$  and  $C_\psi$ , the positive sign is valid for  $\theta \approx -30^\circ$  and the negative sign is valid if  $\theta \approx 30^\circ$ .

## B. Plastic potential derivatives

$$\begin{aligned} \left\{ \frac{\partial Q}{\partial \sigma} \right\} &= \frac{\partial Q}{\partial \sigma_m} \left\{ \frac{\partial \sigma_m}{\partial \sigma} \right\} + \frac{\partial Q}{\partial J_2} \left\{ \frac{\partial J_2}{\partial \sigma} \right\} + \frac{\partial Q}{\partial J_3} \left\{ \frac{\partial J_3}{\partial \sigma} \right\} \\ &= \left( \frac{\partial Q}{\partial \sigma_m} [\mathbf{M}^1] + \frac{\partial Q}{\partial J_2} [\mathbf{M}^2] + \frac{\partial Q}{\partial J_3} [\mathbf{M}^3] \right) \{\sigma\} \end{aligned}$$

where

$$[\mathbf{M}^1] = \frac{1}{3(\sigma_x + \sigma_y + \sigma_z)} \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \text{symmetrical} & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$[\mathbf{M}^2] = \frac{1}{3} \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ 2 & -1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 \\ \text{symmetrical} & 6 & 6 & 6 & 6 & 6 \end{bmatrix}$$

$$[\mathbf{M}^3] = \frac{1}{3} \begin{bmatrix} s_x & s_z & s_y & \tau_{xy} & -2\tau_{yz} & \tau_{zx} \\ s_y & s_x & \tau_{xy} & \tau_{yz} & -2\tau_{zx} & -2\tau_{xy} \\ s_z & -2\tau_{xy} & \tau_{yz} & \tau_{zx} & -2\tau_{xy} & -2\tau_{yz} \\ -3s_z & 3\tau_{zx} & 3\tau_{yz} & 3\tau_{xy} & -3s_x & -3s_y \\ -3s_x & -3s_y & 3\tau_{xy} & -3s_z & -3s_z & -3s_x \\ \text{symmetrical} & & & & & \end{bmatrix}$$

and

$$\{\sigma\} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{Bmatrix}$$

1. Von Mises

$$\frac{\partial Q}{\partial \sigma_m} = 0$$

$$\frac{\partial Q}{\partial J_2} = \sqrt{\frac{3}{2}} \frac{1}{t}$$

$$\frac{\partial Q}{\partial J_3} = 0$$

2. Mohr–Coulomb

$$\frac{\partial Q}{\partial \sigma_m} = \sin \psi$$

$$\frac{\partial Q}{\partial J_2} = \frac{\cos \theta}{\sqrt{2}t} \left( 1 + \tan \theta \tan 3\theta + \frac{\sin \psi}{\sqrt{3}} (\tan 3\theta - \tan \theta) \right)$$

$$\frac{\partial Q}{\partial J_3} = \frac{\sqrt{3} \sin \theta + \sin \psi \cos \theta}{t^2 \cos 3\theta}$$

# Appendix D

## main Library Subprograms

This appendix describes the ‘black box’ and general purpose subprograms (subroutines and functions) held in a library called `main`, which is attached to the main programs described in Chapters 4–11 in this book through the command `USE main`. Subroutines in parentheses are normally called in conjunction with the ones they follow.

All the subprograms can be downloaded from the second author’s website. See [www.mines.edu/~vgriffit/5th\\_ed/Software](http://www.mines.edu/~vgriffit/5th_ed/Software).

### Subprogram descriptions

The following descriptions indicate the `main` library subprograms in alphabetic order, together with the meaning of their arguments. Arguments in **bold** are those returned by the subprograms.

Name	Arguments	Description
<code>bandred</code> (bisect)	<code>a, d, e</code>	Returns the transformed diagonal <b>d</b> and off-diagonal <b>e</b> following transformation of real symmetric band matrix <code>a</code> to tridiagonal form by Jacobi rotations.
<b>bandwidth</b>	<code>g</code>	Returns the maximum bandwidth for an element with steering vector <code>g</code> .
<code>banmul</code>	<code>kb, loads,</code> <b>ans</b>	Returns the product of symmetric band matrix <code>kb</code> stored as a rectangle, and vector <code>loads</code> to give solution vector <b>ans</b> .
<code>bantmul</code>	<code>kb, loads,</code> <b>ans</b>	Returns the product of unsymmetric band matrix <code>kb</code> stored as a rectangle, and vector <code>loads</code> to give solution vector <b>ans</b> .

Name	Arguments	Description
beam_gm	<b>gm</b> , ell	Returns geometric matrix <b>gm</b> for beam element of length <b>ell</b> .
beam_km	<b>km</b> , , ei, ell	Returns stiffness matrix <b>km</b> for beam element of stiffness <b>ei</b> and length <b>ell</b> .
beam_mm	<b>mm</b> , fs, ell	Returns ‘mass’ matrix <b>mm</b> for beam element of $\rho A$ (or foundation stiffness <b>fs</b> ) and length <b>ell</b> .
beemat	<b>bee</b> , deriv	Returns <b>bee</b> matrix for shape function derivatives <b>deriv</b> .
bee8	<b>bee</b> , coord, xi, eta, <b>det</b>	Returns ‘analytical’ form of <b>bee</b> matrix and Jacobian determinant <b>det</b> , for plane 8-node element with nodal coordinates <b>coord</b> at local coordinates <b>xi, eta</b> .
bisect (bandred)	<b>d</b> , e, acheps, <b>ifail</b>	Returns eigenvalues of a tridiagonal matrix whose leading diagonal is <b>d</b> and off-diagonal <b>e</b> . Tolerance is <b>acheps</b> and failure flag <b>ifail</b> . Eigenvalues overwrite <b>d</b> .
bmat_nonaxi	<b>bee</b> , <b>radius</b> , coord, deriv, fun, iflag, lth	Returns <b>bee</b> matrix for axisymmetric bodies subjected to non-axisymmetric loading from shape functions <b>fun</b> and derivatives <b>deriv</b> . <b>radius</b> is the $r$ -coordinate of the Gauss point. <b>lth</b> is the harmonic, and <b>iflag</b> equals 1 for symmetry and $-1$ for anti-symmetry.
checon	loads, oldlds, tol, <b>converged</b>	Returns logical variable <b>converged</b> set to .TRUE. if relative change in <b>loads</b> and <b>oldlds</b> is less than <b>tol</b> .
chobk1 (chobk2)	kb, <b>loads</b>	Forward substitution of Choleski factors held in <b>kb</b> (see Figure 3.20). Result overwrites <b>loads</b> .
chobk2 (chobk1)	kb, <b>loads</b>	Back substitution of Choleski factors, held in <b>kb</b> (see Figure 3.20). Result overwrites <b>loads</b> .
cholin	<b>kb</b>	Factorises matrix <b>kb</b> using Choleski’s method. Factors overwrite <b>kb</b> .

Name	Arguments	Description
comred (comsub)	<b>bk</b>	Factorises complex matrix <b>bk</b> stored as a vector of length neq* (nband+1) using Gaussian [L][U]. Factors overwrite <b>bk</b> .
comsub (comred)	<b>bk</b> , <b>loads</b>	Completes Gaussian forward and back-substitution on complex factors <b>bk</b> stored as a vector of length neq* (nband+1). Result overwrites right-hand-side <b>loads</b> .
contour	loads, g_coord, g_num, ned, argv, nlen, ips	Generates a PostScript contour map to output channel ips. g_coord and g_num hold nodal coordinates and element connectivity. loads holds the nodal values to be contoured. ned holds number of required contour intervals. 4-node quads only. argv holds the file base name and nlen holds the number of characters in argv.
cross_product	b, c, <b>a</b>	Returns matrix <b>a</b> which is the cross-product of column b and row c.
deemat	<b>dee</b> , e, v	Return elastic stress-strain <b>dee</b> matrix in 2D (plane strain) or 3D. e and v are Young's modulus and Poisson's ratio.
<b>determinant</b>	jac	Function returning the determinant of 2D or 3D square matrix jac.
dismsh	loads, nf, ratmax, g_coord, g_num, argv, nlen, ips	Generates a PostScript image of the deformed mesh to output channel. ips. g_coord and g_num hold nodal coordinates and element connectivity. loads holds the nodal displacements. nf holds the nodal freedom array. ratmax holds the ratio of the maximum nodal displacement as plotted as a proportion of the longest x- or y- dimension. argv holds the file base name and nlen holds the number of characters in argv.

Name	Arguments	Description
dismsh_ensi (mesh_ensi)	argv, nlen, step, nf, loads	Outputs a file of displacements in the Ensight Gold format that may be viewed using ParaView (see Section 1.11). Note: files generated by subroutine mesh_ensi are also required. argv holds the file base name and nlen holds the number of characters in argv. loads holds the nodal displacements. nf holds the nodal freedom array and step is a counter that identifies the load step.
ecmat	<b>ecm</b> , fun, ndof, nodof	Returns the consistent mass matrix <b>ecm</b> for an element with shape functions fun, ndof total freedoms and nodof freedoms per node.
elmat	area, rho, <b>emm</b>	Returns the lumped mass matrix <b>emm</b> for a 4- or 8-node plane quadrilateral of area area and density rho.
exc_nods	noexe, exele, g_num, <b>totex</b> ,  <b>ntote, nf</b>	Returns the modified <b>nf</b> array accounting for excavated elements. Updates <b>totex</b> and <b>ntote</b> . noexe holds number of elements to be excavated and exele the element numbers.
fkdiag	<b>kdiag</b> , g	Returns maximum bandwidth <b>kdiag</b> for each row of a skyline storage system from g.
fmacat	vmf1, <b>acat</b>	Returns an intermediate matrix <b>acat</b> from the von Mises flow vector vmf1 as used in Programs 6.6 and 6.7.
fmdsig	<b>dee</b> , e, v	Return elastic stress-strain <b>dee</b> matrix in 2D (plane stress). e and v are Young's modulus and Poisson's ratio.
fmpplat	<b>d2x</b> , <b>d2y</b> , <b>d2xy</b> , points, aa, bb, i	Returns derivative terms <b>d2x</b> , <b>d2y</b> and <b>d2xy</b> for the i-th Gauss point in a rectangular plate element of dimensions aa and bb. points holds the locations of the Gauss points.
fmrmat	vmf1, dsbar, dlam, dee, <b>rmat</b>	Returns matrix <b>rmat</b> from the von Mises flow vector vmf1, invariant dsbar, plastic multiplier dlam and elastic matrix dee as used in equation (6.73).

Name	Arguments	Description
formaa	<code>vmfl, rmat, daatd</code>	Returns modified matrix <b>daatd</b> from the von Mises flow vector <code>vmfl</code> , and matrix <code>rmat</code> as used in equation (6.73).
formkb	<b>kb</b> , km, g	Assembles the lower triangle of a symmetric global stiffness matrix as a rectangle <b>kb</b> of length <code>neq</code> and width <code>nband+1</code> (see Figure 3.20). Constituent element matrices are <code>km</code> . Steering vector <code>g</code> .
formkc	<b>bk</b> , km, cm, g, neq	Assembles the complex upper triangle of a symmetric global stiffness matrix as a vector <b>bk</b> of length <code>neq * (nband+1)</code> . Constituent element matrices are <code>km</code> (real) and <code>cm</code> (imaginary). Steering vector <code>g</code> and number of equations <code>neq</code> .
formke	km, kp, c, <b>ke</b> , theta	Returns the coupled matrix <b>ke</b> from the elastic stiffness matrix <code>km</code> , conductivity matrix <code>kp</code> and coupling matrix <code>c</code> . <code>theta</code> is the scalar time-stepping parameter.
formku	<b>ku</b> , km, g	Returns upper triangular global band matrix <b>ku</b> stored as a rectangle, from symmetric element matrix <code>km</code> and steering vector <code>g</code> .
formlump	<b>diag</b> , emm, g	Returns lumped global mass matrix as a vector <b>diag</b> from consistent element mass matrix <code>emm</code> and steering vector <code>g</code> .
formmm	stress, <b>m1</b> , <b>m2</b> , <b>m3</b>	Returns matrices <b>m1</b> , <b>m2</b> and <b>m3</b> from stresses <code>stress</code> as used in calculation of $\{\partial Q / \partial \sigma\}$ [see equation (6.25)].
formtb	<b>pb</b> , km, g	Returns global full band matrix <b>pb</b> stored as a rectangle from unsymmetric element matrix <code>km</code> and steering vector <code>g</code> .
formupv	<b>ke</b> , c11, c12, c21, c23, c32	Returns unsymmetric element matrix <b>ke</b> from constituent matrices <code>c11, c12, c21, c23</code> and <code>c32</code> for use in $u-p-v$ version of Navier–Stokes equations.

Name	Arguments	Description
form_s	gg,ell,kappa, <b>omega,gamma,s</b>	Returns scalar <b>omega</b> , and vectors <b>gamma</b> and <b>s</b> from array gg, scalar kappa and integer ell in BiCGSTAB.
fsparv	<b>kv</b> , km, g, kdiag	Returns lower triangular global matrix <b>kv</b> stored as a vector in skyline form, from symmetric element matrix km and steering vector g. kdiag holds the locations of the diagonal terms.
gauss_band (solve_band)	<b>pb,work</b>	Returns (Gaussian) factorised unsymmetric full band matrix <b>pb</b> and 'working' array <b>work</b> .
getname	<b>argv,nlen</b>	Returns the base name <b>argv</b> of the data file and its number of characters <b>nlen</b> .
glob_to_axial	<b>axial</b> ,global, coord	Returns axial force <b>axial</b> in 2D or 3D rod element from global force components held in <b>global</b> . Nodal coordinates held in <b>coord</b> .
glob_to_loc	<b>local</b> ,global, gamma,coord	Returns local components <b>local</b> of force and moments from global components held in <b>global</b> . <b>gamma</b> holds element orientation angle (3D only) and <b>coord</b> holds the nodal coordinates. Called by subroutine hinge.
hinge	coord,holdr, action, <b>react</b> ,  prop,iel, etype,gamma	Returns correction reactions <b>react</b> from existing and incremental reactions <b>holdr</b> and <b>action</b> , respectively. <b>coord</b> holds nodal coordinates, <b>prop</b> holds beam properties, <b>iel</b> holds element number, <b>etype</b> holds element type, <b>gamma</b> holds element orientation angle (3D only).
invar	<b>stress,sigm,</b> <b>dsbar,theta</b>	Returns stress invariants <b>sigma,dsbar</b> [equation (6.4)] and Lode angle <b>theta</b> [equation (6.3)], from current stresses held in <b>stress</b> .

Name	Arguments	Description
interp	k,dtim,rt,rl, <b>a1</b> ,nstep	Returns the load/time functions in <b>a1</b> at the calculation time step resolution by linear interpolation. k holds the load/time function number, dtim holds the calculation time step, rt and rl hold the input load/time function and nstep holds the required number of calculation time steps. (Program 11.1)
invert	<b>matrix</b>	Returns the inverse of a small matrix called <b>matrix</b> onto itself.
linmul_sky	kv,disps, <b>loads</b> , kdiag	Returns the product of symmetric matrix kv and vector disps to give solution vector <b>loads</b> . kv is stored as a vector in skyline form, kdiag holds the diagonal locations in kv.
load_function	lf,dtim, <b>a1</b>	Returns the load/time function in <b>a1</b> at the calculation time step resolution by linear interpolation. lf holds the input load/time function and dtim holds the calculation time step. (Chapter 9)
loc_to_glob	local, <b>global</b> , gamma,coord	Returns global components <b>global</b> of force and moments from local components held in local. gamma holds element orientation angle (3D only) and coord holds the nodal coordinates. Called by subroutine hinge.
mcdpl	phi,psi,dee, stress, <b>p1</b>	Returns the plastic stress-strain matrix <b>p1</b> for a Mohr-Coulomb material from the friction angle phi and dilation angle psi (in degrees). stress holds the stresses and dee holds the elastic stress-strain matrix.
mesh	g_coord,g_num, argv,nlen,ips	Generates a PostScript image of the initial (undeformed) mesh to output channel ips. g_coord and g_num hold nodal coordinates and element connectivity. argv holds the file base name and nlen holds the number of characters in argv.

Name	Arguments	Description
mesh_ensi (dismsh_ensi)	argv, nlen, g_coord, g_num, element, etype, nf, loads, step, npri, dtim, solid	Outputs a set of files in the Ensight Gold format that may be viewed using ParaView. g_coord and g_num hold nodal coordinates and element connectivity. argv holds the file base name and nlen holds the number of characters in argv. element is the element type. etype holds element material type. nf holds the nodal freedom array. loads holds applied loads. step is a counter that identifies the load step. npri is the print interval. dtim is the time step size. When solid equals .TRUE., the analysis is for solid mechanics and when .FALSE. it is for fluids. (Section 1.11)
mocouf	phi, c, sigm, dsbar, theta, <b>f</b>	Returns the Mohr–Coulomb failure function <b>f</b> , from the strength parameters phi and c and stress invariants sigm, dsbar and theta.
mocouq	psi, dsbar, theta, <b>dq1</b> , <b>dq2</b> , <b>dq3</b>	Returns the plastic potential terms <b>dq1</b> , <b>dq2</b> and <b>dq3</b> for a Mohr–Coulomb material from dilation angle psi (in degrees) and invariants dsbar and theta [equation (6.25)].
<b>norm</b>	x	Function returns the l2-norm of vector x.
num_to_g	num, nf, <b>g</b>	Returns the element steering vector <b>g</b> from the element node numbering num and the nodal freedom array nf.
pin_jointed	<b>km</b> , ea, coord	Returns the stiffness matrix <b>km</b> of a rod element in 2D or 3D. ea holds the element stiffness and coord holds the element nodal coordinates.
rect_km	<b>km</b> , coord, e, v	Returns the stiffness matrix <b>km</b> of a rectangular plane strain 4- or 8-node quadrilateral element assuming four Gauss points. coord holds the element nodal coordinates, e and v hold Young's modulus and Poisson's ratio, respectively.

Name	Arguments	Description
rigid_jointed	<b>km</b> , prop, gamma, etype, iel, coord	Returns the stiffness matrix <b>km</b> of a beam–rod element in 2D or 3D. coord holds nodal coordinates, prop holds beam properties, iel holds element number, etype holds element type, gamma holds element orientation angle (3D only).
rod_km	<b>km</b> , ea, length	Returns the stiffness matrix <b>km</b> of a rod element in 1D. ea holds the element stiffness and length holds the element length.
rod_mm	<b>mm</b> , length	Returns the mass matrix <b>mm</b> of a rod element. length holds the element length.
sample	element, <b>s</b> , <b>wt</b>	Returns the local coordinates <b>s</b> and weighting coefficients <b>wt</b> for numerical integration of a finite element of type element.
seep4	<b>kp</b> , coord, perm	Returns the ‘analytical’ conductivity matrix <b>kp</b> of a 4-node plane element based on four Gauss points. coord holds the element nodal coordinates and perm holds the permeability matrix.
shape_der	<b>der</b> , points, i	Returns the shape function derivatives <b>der</b> at the i <sup>th</sup> integrating point. points holds the local coordinates of the integrating points.
shape_fun	<b>fun</b> , points, i	Returns the shape functions <b>fun</b> at the i <sup>th</sup> integrating point. points holds the local coordinates of the integrating points.
solve_band (gauss_band)	<b>pb</b> , work, <b>loads</b>	Returns solution <b>loads</b> which overwrites rhs by forward and back-substitution on (Gaussian) factorised unsymmetric full band matrix <b>pb</b> . work holds ‘working’ array.
spabac (sparin)	<b>kv</b> , <b>loads</b> , kdiag	Returns solution <b>loads</b> which overwrites rhs by forward and back-substitution on (Choleski) factorised vector <b>kv</b> stored as a skyline. kdiag holds the locations of the diagonal terms.

Name	Arguments	Description
spabac_gauss (sparin_gauss)	<b>kv</b> , <b>loads</b> , kdiag	Returns solution <b>loads</b> which overwrites rhs by forward and back-substitution on (Gaussian) factorised vector <b>kv</b> stored as a skyline. kdiag holds the locations of the diagonal terms.
sparin (spabac)	<b>kv</b> , kdiag	Returns the (Choleski) factorised vector <b>kv</b> stored as a skyline. kdiag holds the locations of the diagonal terms.
sparin_gauss (spabac_gauss)	<b>kv</b> , kdiag	Returns the (Gaussian) factorised vector <b>kv</b> stored as a skyline. kdiag holds the locations of the diagonal terms.
stability	<b>kv</b> , <b>gv</b> , kdiag, tol, limit, iters, <b>evec</b> , <b>eval</b>	Returns the smallest eigenvalue <b>eval</b> and corresponding eigenvector <b>evec</b> of a compressed beam with global stiffness and geometric matrices held in <b>kv</b> and <b>gv</b> , respectively. kdiag holds the locations of the diagonal terms, limit is the iteration ceiling and iters is the number of iterations to reach convergence with tolerance tol.
stiff3 stiff6 stiff10 stiff15	<b>km</b> , coord, ym, pr	Returns the exact stiffness matrix <b>km</b> of a plane strain 3, 6, 10 or 15-noded straight-sided triangular element. coord holds the element nodal coordinates, ym and pr hold Young's modulus and Poisson's ratio, respectively.
stiff4	<b>km</b> , coord, ym, pr	Returns the stiffness matrix <b>km</b> of a general plane strain 4-node quadrilateral element based on four Gauss points. coord holds the element nodal coordinates, ym and pr hold Young's modulus and Poisson's ratio, respectively.

---

Name	Arguments	Description
vecmsh	<code>loads, nf, ratmax, cutoff, g_coord, g_num, argv, nlen, ips</code>	Generates a PostScript image of the nodal displacement vectors to output channel <code>ips</code> . <code>g_coord</code> and <code>g_num</code> hold nodal coordinates and element connectivity. <code>loads</code> holds the nodal displacements. <code>nf</code> holds the nodal freedom array. <code>ratmax</code> holds the ratio of the maximum vector length plotted as a proportion of the longest $x$ - or $y$ -dimension. <code>cutoff</code> gives the length of the shortest vector to be plotted as a proportion of the longest. <code>argv</code> holds the file base name and <code>nlen</code> holds the number of characters in <code>argv</code> .
vmdpl	<code>dee, stress, p1</code>	Returns the plastic stress-strain matrix <code>p1</code> for a von Mises material. <code>stress</code> holds the stresses and <code>dee</code> holds the elastic stress-strain matrix.
vmflow	<code>stress, dsbar, vmf1</code>	Returns the von Mises flow vector <code>vmf1</code> . <code>stress</code> holds the stresses and <code>dsbar</code> holds the second deviatoric invariant.

---



# Appendix E

## geom Library Subroutines

This appendix describes the ‘geometry’ subroutines held in a library called `geom`, which is attached to the main programs described in Chapters 4–11 in this book through the command `USE geom`.

All the subroutines can be downloaded from the second author’s website. See [www.mines.edu/~vgriffit/5th\\_ed/Software](http://www.mines.edu/~vgriffit/5th_ed/Software).

### Subroutine descriptions

The following descriptions indicate the `geom` library subroutines in alphabetic order, together with the meaning of their arguments. Arguments in **bold** are those returned by the subroutine.

Name	Arguments	Description
<code>bc_rect</code>	<code>nxe,nye,nf,dir</code>	Returns nodal freedom array <b>nf</b> for a rectangular 2D mesh of 8-node quadrilaterals with <code>nxe</code> columns and <code>nye</code> rows of elements. Nodes numbered in the <code>dir</code> direction. Vertical rollers on sides, fully fixed on base
<code>emb_2d_bc</code>	<code>nx1,nx2,ny1,</code> <code>ny2,nf</code>	Returns nodal freedom array <b>nf</b> for a 2D embankment analysis with mesh defined by <code>nx1,nx2,ny1</code> and <code>ny2</code>
<code>emb_2d_geom</code>	<code>iel,nx1,nx2,</code> <code>ny1,ny2,w1,</code> <code>s1,w2,h1,h2,</code> <b>coord, num</b>	Returns element nodal coordinates <b>coord</b> and node numbers <b>num</b> for a 2D embankment analysis with geometry and mesh defined by <code>nx1,nx2,ny1,ny2</code> , <code>w1,s1,w2,h1,h2</code>

Name	Arguments	Description
emb_3d_bc	ifix, nx1, nx2, ny1, ny2, nze, <b>nf</b>	Returns nodal freedom array <b>nf</b> for a 3D embankment analysis with mesh defined by nx1, nx2, ny1, ny2, nze. ifix defines boundary conditions.
emb_3d_geom	iel, nx1, nx2, ny1, ny2, nze, w1, s1, w2, h1, h2, d1, <b>coord</b> , <b>num</b>	Returns element nodal coordinates <b>coord</b> and node numbers <b>num</b> for a 3D embankment analysis with geometry and mesh defined by nx1, nx2, ny1, ny2, nze, w1, s1, w2, h1, h2, d1
fmcoem	g_num, <b>g_coord</b> , fwidth, fdepth, width, depth, lnxe, lifts, fnxe, fnye, itype	Returns the global coordinates <b>g_coord</b> for a sequential embankment analysis. g_num holds the global node numbering. Mesh coordinates fwidth, fdepth and width, depth. Number of columns of <b>element</b> in embankment lnxe. Foundation discretisation fnxe, fnye. Number of lifts lifts. Slope element type itype.
fmglem	fnxe, fnye, lnxe, <b>g_num</b> , lifts	Returns the global node numbering <b>g_num</b> for a sequential embankment analysis. Foundation discretisation fnxe, fnye. Number of columns of elements in embankment lnxe. Number of lifts lifts.
formnf	<b>nf</b>	Returns nodal freedom array <b>nf</b> from boundary conditions input of 0s and 1s.
geom_freesurf	iel, nxe, fixed_seep, fixed_down, down, width, angs, surf, <b>coord</b> , <b>num</b>	Returns element nodal coordinates <b>coord</b> and node numbers <b>num</b> for a 2D freesurface analysis. iel is the element number. nxe is the number of columns of elements. fixed_seep is the number of nodes on the seepage surface. fixed_down and down are the number of nodes and the fixed head on the downstream side. width and angs are the x-coordinates and inclination angle at the base of the mesh. surf holds the y-coordinates of the freesurface.

---

Name	Arguments	Description
geom_rect	element, iel, x_coords, y_coords, <b>coord, num</b> , dir	Returns element nodal coordinates <b>coord</b> and node numbers <b>num</b> for a rectangular mesh of triangles or quadrilaterals. Element type is <b>element</b> . <b>iel</b> is the element number. <b>x_coords</b> and <b>y_coords</b> are the <b>x</b> - and <b>y</b> - coordinates of the mesh. <b>dir</b> is the node numbering direction.
hexahedron_xz	iel, x_coords, y_coords, z_coords, <b>coord, num</b>	Returns element nodal coordinates <b>coord</b> and node numbers <b>num</b> for a cuboidal mesh of hexahedrons with nodes numbered in <b>x</b> - then <b>z</b> - then <b>y</b> . <b>iel</b> is the element number.
mesh_size	element, nod, <b>nels, nn</b> , nxe, nye, nze	Returns the number of elements <b>nels</b> and the number of nodes <b>nn</b> in a rectangular mesh of triangles or quadrilaterals. Element type is <b>element</b> with <b>nod</b> nodes.

---



# Appendix F

## Parallel Library Subroutines

This appendix describes the additional library subroutines used by the Chapter 12 programs illustrating parallel finite element computations. All the subroutines can be downloaded from <http://parafem.org.uk> or [www.mines.edu/~vgriffit/5th\\_ed/Software](http://www.mines.edu/~vgriffit/5th_ed/Software).

### Subroutine descriptions

The following descriptions indicate the library subroutines in alphabetic order, together with the meaning of their arguments. Arguments in **bold** are those returned by the subroutine.

Name	Arguments	Description
abaqus2sg	element, <b>g_num_pp</b>	Converts element connectivity <b>g_num_pp</b> from the ABAQUS convention to the book convention. element type can be ‘hexahedron’ or ‘tetrahedron’.
biot_cube_bc20	nxe, nye, nze, <b>rest</b>	Returns restraint array <b>rest</b> for numbers of elements in x, y, z of 20/8-node bricks.
biot_loading	nxe, nze, nle, <b>no, val</b>	Returns loaded freedoms <b>no</b> and values <b>val</b> from number of elements in x and z and number of loaded elements along a side of a square.
box_bc8	nxe, nye, nze, <b>rest</b>	Returns restraint array <b>rest</b> for numbers of elements in x, y, z of 8-node bricks.

Name	Arguments	Description
calc_nels_pp	argv, nels, npes, partitioner, <b>nels_pp</b>	Returns number of elements <b>nels_pp</b> on each parallel process. If partitioner equals 1, <b>nels_pp</b> is calculated by dividing nels by npes. If partitioner equals 2, an external tool has been used and <b>nels_pp</b> is read in from file argv.pszie.
calc_neq_pp	None	Returns number of equations on each parallel process.
calc_nodes_pp	nn, npes, numpe, <b>node_end</b> , <b>node_start</b> , <b>nodes_pp</b>	Returns number of nodes <b>nodes_pp</b> , the first node number <b>node_start</b> and last node number <b>node_end</b> on each process. numpe is the ID number of the process. nn is the number of nodes. npes is the total number of processes used in the analysis.
calc_npes_pp	npes, <b>npes_pp</b>	Returns number of processes <b>npes_pp</b> each process communicates with. npes is the total number of processes used in the analysis.
checon_par	xnew_pp, tol, <b>converged</b> , x_pp	Parallel version of checon. <b>converged</b> returned as .TRUE. if difference between xnew_pp and x_pp less than tol.
cube_bc20	nxe, nye, nze, <b>rest</b>	Returns restraint array <b>rest</b> for numbers of elements in x, y, z of 20-node bricks.
dismsh_ensi_p	unit, iy, nodes_pp, npes, numpe, j, temp	Collects and writes nodal results temp to a file identified by its unit number. numpe is the process ID. nodes_pp is the number of nodes per process. iy is a load increment counter. j is the number of components being written.
dismsh_pb	unit, iy, nodes_pp, npes, numpe, j, temp	Collects and writes nodal results temp in BINARY format to a file identified by its unit number. numpe is the process ID. nodes_pp is the number of nodes per process. iy is a load increment counter. j is the number of components being written.

Name	Arguments	Description
<b>DOT__PRODUCT__P</b>	a_pp, b_pp	Returns the dot product of distributed vectors a_pp and b_pp to all processes. Parallel equivalent of FORTRAN intrinsic DOT_PRODUCT.
find_g find_g4 find_g3	num, g, rest	Returns steering vector <b>g</b> from nodes num and restraints rest.
find_no	node, rest, sense, no	Returns vector of fixed freedom numbers <b>no</b> from vector of fixed nodes node, restraints rest and sense of freedoms to be fixed.
find_no2	g_g_pp, g_num_pp node, sense, no	Returns vector of fixed freedom numbers <b>no</b> from vector of fixed nodes node, and sense of freedoms to be fixed. g_g_pp is the distributed steering vector. g_num_pp is the distributed element connectivity.
find_pe_procs	<b>numpe, npes</b>	Returns number of MPI processes <b>npes</b> being used and the ID number or rank <b>numpe</b> of each process.
formupvw	<b>storke_pp</b> ,iel, c11,c12,c21, c23,c32,c24, c42	Returns the element ke matrix for 3D Navier–Stokes from constituent matrices c11, c12, c21, c23, c32, c24, c42 and stores it in array <b>storke_pp</b> . iel is a simple element counter.
gather	p_pp, <b>pmul_pp</b>	Gathers distributed p_pp into <b>pmul_pp</b> .
geometry_8bxz	ielpe,nxe,nze, aa,bb,cc, <b>coord, num</b>	Returns element connectivity <b>num</b> and nodal coordinates <b>coord</b> for a box of 8-node bricks with nxe elements in x and nze elements in z with x, y, z sizes aa, bb, cc. ielpe is the first element number on the process.
geometry_20bxz	ielpe,nxe,nze, aa,bb,cc, <b>coord, num</b>	Returns element connectivity <b>num</b> and nodal coordinates <b>coord</b> for a box of 20-node bricks with nxe elements in x and nze elements in z with x, y, z sizes aa, bb, cc. ielpe is the first element number on the process.

Name	Arguments	Description
g_t_g	nod, g_t, <b>g</b>	Returns steering vector <b>g</b> (Biot) from total vector g_t. nod is the number of nodes per element.
g_t_g_ns	nod, g_t, <b>g</b>	Returns steering vector <b>g</b> (Navier-Stokes) from total vector g_t. nod is the number of nodes per element.
load	g_g_pp, g_num_pp, node, val, <b>fext_pp</b>	Returns distributed loads <b>fext_pp</b> from nodes to be loaded node and prescribed load values val. g_g_pp are the distributed global steering vectors. g_num_pp are distributed element connectivities.
loading	nxe, nze, nle, <b>no, val</b>	Returns loaded freedoms <b>no</b> and values <b>val</b> from number of elements in x and z and number of loaded elements along a side of a square.
make_ggl	npes_pp, npes, g_g_pp	Builds inter-process communication tables for gather/scatter using the distributed steering array g_g_pp. npes_pp is the number of processes each process must communicate with and npes is the total number of processes used in the analysis.
max_p	<b>scalar</b> <b>integer</b>	Function returns the global max of a <b>scalar integer</b> distributed across processes.
ns_cube_bc20	nxe, nye, nze, <b>rest</b>	Returns restraint array <b>rest</b> for numbers of elements in x, y, z of 20/8-node bricks.
ns_loading	nxe, nye, nze, <b>no</b>	Returns loaded freedoms <b>no</b> for a cuboid with nxe, nye, nze elements in x, y, z.
read_g_coord_pp	argv, g_num_pp, nn, npes, numpe, <b>g_coord_pp</b>	First process reads global coordinates from file argv.d and distributes them to <b>g_coord_pp</b> . g_num_pp, is the distributed element connectivity matrix. nn is the number of nodes. numpe is the process ID. npes is the total number of processes used in the analysis.

Name	Arguments	Description
read_fixed	argv, numpe, <b>nodef, sense,</b> <b>val_f</b>	First process reads <b>nodef</b> nodes to be fixed, <b>sense</b> of freedom to be fixed and <b>val_f</b> values from file argv.fix, then broadcasts to other processes. numpe is the process ID.
read_g_num_pp	argv, iel_start, nn, npes, numpe, <b>g_num_pp</b>	First process reads element connectivities from file argv.d and distributes them to <b>g_num_pp</b> . iel_start is the first element on the process. nn is the number of nodes. npes is the total number of processes. numpe is the process ID.
read_material	argv, <b>prop</b> , numpe, npes	First process reads material property values <b>prop</b> from file argv.mat and broadcasts them to the other processes. numpe is the process ID. npes is the total number of processes used in the analysis.
read_p???	REAL and INTEGER data to be broadcast	First process reads program ‘control’ data and broadcasts it to all the other processes. A separate subroutine is provided for Programs 12.1 to 12.10.
read_qinc	argv, numpe, <b>qinc</b>	First process reads vector of load increment terms <b>qinc</b> from file argv.dat and broadcasts it to all the other processes. numpe is the process ID.
read_rest	argv, numpe, <b>rest</b>	First process reads node freedom restraints <b>rest</b> from file argv.bnd and broadcasts it to all the other processes. numpe is the process ID.
rearrange rearrange_2	<b>rest</b>	Reorganises restraint array <b>rest</b> .
reindex	ieq_start,no, <b>no_pp_temp</b> , <b>fixed_</b> <b>freedoms_pp</b> , <b>fixed_</b> <b>freedoms_start</b> , <b>neq_pp</b>	Locates loaded or displaced freedoms in no on each process and stores them in <b>no_pp_temp</b> . <b>fixed_freedoms_pp</b> is the number of fixed freedoms on the process. <b>fixed_freedoms_start</b> is the first fixed freedom on the process. <b>neq_pp</b> is the number of equations per process.

Name	Arguments	Description
scatter	<b>u_pp</b> , utemp_pp	Scatters utemp_pp to distributed vector <b>u_pp</b> .
scatter_nodes	npes, nn, nels_pp, g_num_pp, nod, ndim, nodes_pp, node_start, node_end, eld_pp, <b>disp_pp</b> , flag	Scatters nodal values eld_pp to distributed vector <b>disp_pp</b> . npes is the number of processes. nn is the number of nodes. nels_pp is the number of elements per process. g_num_pp holds the distributed element connectivities. nod is the number of nodes per element. ndim is the number of dimensions. nodes_pp is the number of nodes per process. node_start is the first node and node_end is the last node of the process. If flag equals 1, nodal averaging is applied.
scatter_nodes_ns	npes, nn, nels_pp, g_num_pp, nod, nodof, nodes_pp, node_start, node_end, eld_pp, <b>upvw_pp</b> , flag	Scatters nodal values eld_pp to distributed vector <b>upvw_pp</b> . npes is the number of processes. nn is the number of nodes. nels is the number of elements per process. g_num_pp holds the distributed element connectivities. nod is the number of nodes per element. nodof is the number of degrees of freedom. nodes_pp is the number of nodes per process. node_start is the first node and node_end is the last node of the process. If flag equals 1, nodal averaging is applied (Navier-Stokes).
SHUTDOWN	None	Finalises MPI and calls STOP.
SUM_P	<b>u_pp</b>	Returns vector sum of distributed vector <b>u_pp</b> .
vmp1	dee, stress, <b>p1</b>	See vmdpl, Appendix D.

# Appendix G

## External Subprograms

This appendix describes several external subprograms (subroutines and functions) used by some of the main programs in this book, but not provided directly by the authors. Interested readers can request the programs from the developers as indicated in the list below.

### Subprogram descriptions

The following descriptions list the external subprograms in alphabetic order, together with the meaning of their arguments. Arguments in **bold** are those returned by the subprograms.

Name	Arguments	Description
daxpy	<code>n, da, dx, incx, <b>dy</b>, incy</code>	Double precision basic linear algebra subprogram (BLAS) for multiplying vector <code>dx</code> of size <code>n</code> by scalar real <code>da</code> , returning the result to vector <code>dy</code> . See <a href="http://www.netlib.orgblas/daxpy.f">http://www.netlib.orgblas/daxpy.f</a> for further details regarding the arguments.
<b>ddot</b>	<code>n, dx, incx, dy, incy</code>	Double precision basic linear algebra subprogram (BLAS) of type ‘function’ for computing the dot product ‘ddot’ of vectors <code>dx</code> and <code>dy</code> of size <code>n</code> . See <a href="http://www.netlib.orgblas/ddot.f">http://www.netlib.orgblas/ddot.f</a> for further details regarding the arguments.

Name	Arguments	Description
dgemv	<code>trans,m,n, alpha,a,lda, x,incx,beta, y,incy</code>	Double precision basic linear algebra subprogram (BLAS) which performs one of the matrix–vector operations $y = \alpha * a * x + \beta * y$ or $y = \alpha * a * x$ where $\alpha$ and $\beta$ are scalars, $x$ and $y$ are vectors and $a$ is an $m$ by $n$ matrix. See <a href="http://www.netlib.orgblas/dgemv.f">http://www.netlib.orgblas/dgemv.f</a> for further details regarding the arguments.
dsaupd	<code>ido,bmat,neq, which,nev,tol, resid,ncv,v, neq,iparam, ipntr,workd, workl,lworkl, info</code>	Reverse communication interface for the implicitly restarted Arnoldi iteration (ARPACK). See <a href="http://www.caam.rice.edu/software/ARPACK/UG/">http://www.caam.rice.edu/software/ARPACK/UG/</a> for further details regarding the arguments.
dseupd	<code>rvec,'All', select d, v,neq,sigma, bmat,neq,which, nev,tol,resid, ncv, v, neq,iparam, ipntr,workd, workl,lworkl, ierr</code>	Extracts computed eigenvalues and eigenvectors for the implicitly restarted Arnoldi iteration (ARPACK). See <a href="http://www.caam.rice.edu/software/ARPACK/UG/">http://www.caam.rice.edu/software/ARPACK/UG/</a> for further details regarding the arguments.
dsymm	<code>side,uplo,m,n, alpha,a,lda,b, ldb,beta,c, ldc</code>	Double precision basic linear algebra subprogram (BLAS) which performs one of the matrix–matrix operations $c = \alpha * a * b + \beta * c$ , or $c=\alpha * b * a + \beta * c$ , where $\alpha$ and $\beta$ are scalars, $a$ is a symmetric matrix and $b$ and $c$ are $m$ by $n$ matrices. See <a href="http://www.netlib.orgblas/dsymm.f">http://www.netlib.orgblas/dsymm.f</a> for further details
lancz1 (lancz2)	<code>n,el,er,acc, leig,lx,lalpha, lp,iflag, u,v,eig,jeig, neig,x,del,nu, alfa,beta, v_store</code>	Lanczos method for eigenvalues. See Chapter 10 and website reference HSL (2011) for details. Alias for ea25a/ad.

---

Name	Arguments	Description
lancz2 (lancz1)	n, lalpha, lp, eig, jeig, neig, alfa, beta, lz, <b>jflag, y,</b> w, z, v_store	Lanczos method for eigenvectors. See Chapter 10 and website reference HSL (2011) for details. Alias for ea25e/ed.
umat	<b>sigma, statev, dee,</b> sse, spd, scd, rpl, ddsddt, drplde, drpldt, stran, eld, time, dtime, temp, dtemp, predef, dpred, cmname, ndi, nshrv, nst, nstatv, props, nprops, points, drot, pnewdt, celent, dfgrd0, dfgrd1, iel, npt, layer, kspt, kstep, kinc	Abaqus user material subroutine. Only the arguments sigma and dee are outputs in the Program 5.7. All of the arguments listed are part of the ABAQUS UMAT interface, but most are not used in the Program 5.7 example.  See Abaqus documentation at <a href="http://www.3ds.com/products/simulia/support/documentation/">http://www.3ds.com/products/simulia/support/documentation/</a>

---



# Author Index

- Agullo E 17  
Ahmad S 61  
Arnoldi WE 71
- Bai Z 71–2  
Bates KT 548  
Bathe KJ 47, 67, 76, 485  
Beer G 10  
Belytschko T 279, 289  
Berg PN 54  
Biot MA 54, 423  
Bishop AW 264–5  
Brassey CA 561
- Cardoso JP 63  
Carslaw HS 384–6, 390, 400–401  
Chan KH 71, 571  
Chan SH 71, 571  
Chang CY 253  
Chopra AK 465  
Christias D 575  
Cook RD 25, 464  
Cormeau IC 240–241  
Crisfield MA 289  
Cuthill E 175
- Davis EH 177  
Demmel J 17, 71–2  
der Vorst HV 71–2  
DeWitt DP 413, 415  
Dijkstra EW 15  
Dobbins WE 405, 407
- Dongarra J 17, 71–2  
Dongarra JJ 17  
Dudarev S 558  
Duenser C 10  
Duncan JM 235  
Dunne F 217
- Ergatoudis J 61  
Evans LL 558
- Falkingham PF 548  
Farraday RV 54, 408  
Fenton GA 594  
Finlayson BA 27  
Fix GJ 25  
Fokkema DR 70–71
- Gere JM 153  
Gilvary B 70, 103  
Gladwell I 70, 101, 103, 510  
Gladwell I 103  
Goodier JN 36, 39, 41, 42  
Greenbaum A 70  
Griffiths DV 27, 33, 53, 55, 59, 63, 65,  
68–9, 71, 82, 106, 144, 188, 242,  
244, 263, 279, 308, 312–13, 341,  
343, 376, 423, 594, 619
- Hadri B 17  
Heshmati EE 473  
Hetenyi M 133  
Hicks MA 106  
Hill R 235  
Ho DKH 297

- Hobbs R 54, 290, 443  
Horne MR 32, 145, 148, 168  
Huang J 53, 279, 376, 436  
Hughes TG 50  
Hughes TJR 70, 101–2, 244  
Humpheson C 244, 262  
Hwang R 111  
  
Idriss IM 111  
Incropera FP 413, 415  
Irons BM 62, 193  
  
Jaeger JC 384–6, 390, 400–401  
Jennings A 47  
Johns DJ 209  
  
Karypis G 534  
Kelley CT 70  
Key SW 505  
Kidger DJ 88–90  
King IP 242  
Kitchener AC 561  
Kopal A 62  
Krieg OB 289  
Krieg RD 289  
Kumar V 534  
Kurzak J 17  
  
Lambe TW 459  
Lane PA 263  
Langou J 17  
Leckie FA 31  
Lee GC 38  
Lee FH 71  
Lehoucq RB 71  
Levit I 70, 101–2  
Lewis RW 244, 262  
Li L 571  
Liao X 571  
Ligang L 571  
Lindberg GM 31  
Lindsey CH 15  
Liu WK 279, 289  
Livesley RK 32  
Lysmer J 112  
  
Malkus DS 25, 464  
Manning PL 548, 561  
  
Mar A 172  
Margetts L 534, 571  
Martin CM 259  
McKee J 175  
McKeown JJ 47, 68  
Merchant W 32  
Molenkamp F 235  
Moran B 279, 289  
Morgenstern NR 264  
Mummery PM 558  
Muskat M 52  
Mustoe GGW 244  
  
Nayak GC 270  
Naylor DJ 306  
  
O'Connor BA 54, 408  
Ortiz M 272  
  
Parlett BN 98, 472  
Peano A 202  
Petrinic N 217  
Pettipher MA 5, 523  
Phoon KK 71  
Plesha ME 25, 464  
Popov EP 272  
Poulos HG 177  
Przemieniecki JS 130  
  
Rao SS 25  
Reid JK 98  
Rice JR 274  
Ruhe A 71–2  
  
Sakurai T 242  
Sandhu RS 106  
Schiermeyer RP 65  
Schiffman RL 375, 377, 443  
Schlichting H 48  
Scott FC 61  
Seed HB 111  
Sellers WI 561  
Simo JC 279  
Sleijpen GLG 70–71  
Smith IM 4–5, 27, 51, 54, 68–71, 74,  
88–90, 101, 113, 153, 200, 233,  
290, 297, 408, 410, 443, 468, 473,  
506, 523, 534, 540, 571

- Sorensen DC 71  
Stein JR 376– 7  
Strang G 25  
Szabo BA 38
- Taig IC 37, 59  
Taylor C 50  
Taylor DW 279, 294, 297, 306, 373,  
  375  
Taylor RL 25, 63–4, 186, 241  
Timoshenko SP 36, 40–41, 43, 45, 107,  
  112, 153, 156  
Too J 63–4  
Tracey DM 274
- Udaka T 112
- Valliappan S 242  
van der Vorst HA 70–71  
Verruijt A 348
- Walker DW 17  
Wang A 540
- Warburton GB 487–8  
Weaver W 107, 112  
Whitman RV 459  
Willé DR 2  
Willson SM 242, 619  
Wilson EL 47, 67, 76, 106, 186  
Winget J 70, 101–2  
Withers PJ 561  
Witt RJ 25, 464  
Woinowsky-Krieger S 44, 156  
Wong SW 113, 510
- Xinhao L 571
- Yamada Y 242, 619  
Yang C 71  
Yoshimura N 242  
Young D 107, 112  
Young P 558
- Zhang K 571  
Zhu JZ 25, 186, 241  
Zienkiewicz OC 25, 61, 63–64, 186,  
  240–242, 244, 262, 270



# Subject Index

- ABAQUS 18, 213, 216–17, 221–3, 639, 647  
advection terms 53  
analysis  
    axially loaded elastic rods 1D 116–21  
    Biot poro-elastic solid 3D 571–6  
    elastic beams 127–33  
    elastic pin-jointed frames 121–6  
    elastic rigid-jointed frames 133–49  
    elastic solid 3D 196–205  
    elasto-plastic (Mohr–Coulomb) solid 542–8  
    plane free-surface flow 344–51  
    plane steady state Navier–Stokes equation 424–9  
    plane steady state Navier–Stokes equation (element-by-element solution) 429–33  
    plates using 4-node rectangular plate elements 153–7  
    steady seepage 1D 334–7  
applications software 5–9  
Arnoldi method 71  
ARPACK 17, 71, 99, 474–80, 646  
arrays 9–16  
    computation functions 11  
    dynamic arrays 9  
    inspection functions 11  
    intrinsic procedures 11–12  
    sections referencing 11  
    whole-array manipulations 11  
aspect ratio 38  
assembly subroutines 77t, 95t, 97f  
axisymmetric analysis, non-rectangular elements 342  
axisymmetric elastic solids,  
    non-axisymmetric analysis 184–91  
axisymmetric elements (2D) 607–9  
axisymmetric foundation analysis 182  
axisymmetric strain 40–42  
    degrees of freedom 94  
    of elastic solids 82  
    elastic–plastic solid “undrained” 308–13  
axisymmetric stress 40–42  
backward Euler method 274–5  
bandred (main library) 76, 623  
bandwidth (main library) 623  
bandwidth optimisers 175  
banmul (main library) 623  
bantmul (main library) 623  
beam\_ge (main library) 624  
beam\_km (main library) 624  
beam\_mm (main library) 624  
beam analysis, nodal loading 129  
beam–column elements 30  
beam elements, 29–31  
    mass matrix 31  
    node and freedom numbering 129f  
    slender 29f  
    stiffness matrix 29–30  
beam geometric matrix 32  
beam–rod elements

- node and freedom numbering (2D)  
     137f  
 node and freedom numbering (3D)  
     139f  
 stiffness matrix 135  
 beams  
     2-node beams 611  
     with axial forces 31–2  
     on elastic foundations 32–3  
 bee8 (main library) 64, 624  
 beemat (main library) 175, 624  
 bending moment 131  
 bent plate, strain energy 44  
 BiCGStab (Stabilised bi-conjugate gradient) 70  
 BiCGStab(I) (Stabilised hybrid bi-conjugate gradient) 70–71, 104, 423, 429  
 biot\_cube\_bc20 (parallel library) 639  
 biot\_loading (parallel library) 639  
 Biot poro-elastic solids  
     3D analysis 571–6  
     plane strain consolidation analysis  
         (absolute load version) 448–54  
     plane strain consolidation analysis  
         (incremental version) 438–44  
 Biot poro-elastic–plastic materials, plane strain consolidation analysis 448–54  
 Biot’s equations for coupled consolidation 71  
 Biot’s theory of coupled solid–fluid interaction 54  
 bisect (bandred) (main library) 76, 624  
 black box routines 76–7  
 BLAS (Basic Linear Algebra Subprogram) libraries 17–18, 76, 221, 592, 645–6  
 bmat\_nonaxi (main library) 624  
 body loads, generation of 240  
 body loads vector 268  
 boundary conditions 72–5  
     fixed potential 339  
     free surface flow 346f  
     problem-specific routines 527  
 box\_bc8 (parallel library) 639  
 brick element (20-node), stiffness matrix 88  
 broadcasting 9  
 buckling load 151, 160  
 calc\_nels\_pp (parallel library) 640  
 calc\_neq\_pp (parallel library) 640  
 Cartesian stress tensor 236  
 character variables, glossary 456  
 checon (main library) 346, 624  
 checon\_par (parallel library) 640  
 Cholesky factorisation 98  
 cloud computing 2, 523, 594–6  
 coefficient of consolidation 373  
 compilers 5–6  
 complex response method 112, 483, 502  
 computer strategies 1–24  
 conductivity matrix, numerical integration 83  
 conjugate gradient method 68–9  
 consistent mass approximations 98  
 consistent tangent matrix 275  
 consolidation analysis (1D), 2-node “line” elements 370–373  
 consolidation equation  
     general 2-(plane) or 3D analysis 397–401  
     plane or axisymmetric analysis 386–90  
 constant stiffness iterations 233, 234f, 240, 253  
 constructors 9–11  
 contour (main library) 338, 382, 398, 625  
 contour map  
     excess pore pressure 382, 386f  
     nodal potential 341, 342f  
 convection boundary conditions 74–5, 75f, 369, 412, 415–16, 418  
 convergence criterion 276–289  
 coupled Navier–Stokes problems, solution of 103–4  
 coupled problems 423–59  
     exercises 459  
     glossary 454–8  
     programs 423–59  
 coupled solid–fluid problems (3D), degrees of freedom per node 94  
 coupled transient problems, solution of 104–6

- Crank–Nicolson method of time integration 99, 109–10, 371, 453, 485  
`cross_product` (main library) 76, 83, 625  
`cube_bc20` (parallel library) 640
- dams  
embankment free surface analysis configuration and mesh 350f  
flow of water through 346  
sloping sides 350  
vertical face dam analysis 349f  
vertical-sided 347
- `deemat` (main library) 625
- determinant (main library) 76, 625
- deviator stress 313
- diakoptics 533
- diffusion–convection equation 51  
plane analysis (self-adjoint transformation) 401–5  
plane analysis (untransformed solution) 405–10
- diffusion equation 51
- dilation angle 305
- dinosaur tracks 546, 548f
- direct Newmark method 485
- `dismsh` (main library) 625
- displacement vectors 249–50, 259, 271f, 294, 297f, 305, 307f
- distributed arrays, `_pp` appendage 527
- distributed memory systems 5
- domain composition, parallel and serial programs 533–4
- Dupuit formula 348
- dynamic character arrays, glossary 520
- dynamic integer arrays, glossary 158, 222, 325, 360, 417, 456–7, 479, 519, 600
- dynamic real arrays, glossary 158–9, 223–4, 326–7, 360–361, 417–18, 457–8, 479–80, 519–20, 600–602
- `ecmat` (main library) 495, 626
- effective stress 435
- eigenvalue analysis  
3D elastic solids 576–81  
elastic beams 462–5
- eigenvalue analysis of an elastic solid using 4- or 8-node rectangular quadrilaterals 465–9  
using 4-node rectangular quadrilaterals (consistent mass) 469–74  
using 4-node rectangular quadrilaterals (lumped mass) 474–7
- eigenvalue equation 28
- eigenvalue problems 47, 68, 72, 90, 461–77  
exercises 480–482  
glossary 477–80  
programs 462–77
- eigenvalues, evaluation of 96–9
- eigenvectors, evaluation of 96–9
- elastic beams  
analysis 127–33  
eigenvalue analysis 462–5  
forced vibration analysis 484–8  
stability analysis 150–153
- elastic cubes, analysis 540
- elastic–perfectly plastic stress–strain law 235f
- elastic pin-jointed frames analysis 121–6
- elastic–plastic embankments, plane strain construction 276–83
- elastic–plastic excavation, plane strain construction 290–294
- elastic–plastic materials  
plane strain bearing capacity analysis 244–54, 277–81
- plane strain bearing capacity analysis (no global stiffness matrix assembly) 282–6
- plane strain bearing capacity analysis (viscoplastic strain method) 254–60
- plane strain earth pressure analysis 265–70
- plane strain slope stability analysis 260–265
- elastic–plastic slopes  
3D strain analysis 319–22  
viscoplastic strain method analysis 313–18
- elastic–plastic solids, axisymmetric “undrained” strain 308–13

- elastic–plastic (von Mises) solid in plane strain, forced vibration analysis 512–17
- elastic rigid-jointed frames analysis 133–41
- elastic rod element stiffness matrix 117
- elastic rods, 1D analysis 116–21
- elastic solid in plane strain
- eigenvalue analysis 465–9
  - eigenvalue analysis (consistent mass) 469–74
  - eigenvalue analysis (lumped mass) 465–9
  - forced vibration analysis 489–93, 501–8
  - forced vibration analysis (theta method) 493–7, 508–12
  - forced vibration analysis (Wilson’s method) 498–501
- elastic solids
- 2-(plane strain) or 3D analysis 196–205
  - 3D analysis 196–205
  - 3D strain 213–21
  - axisymmetric strain 82
  - eigenvalue analysis 576–81
  - element stiffness 88
  - forced vibration analysis 581–5
  - plane or axisymmetric strain analysis 170–84
- elastic stress–strain matrix (3D) 44
- elasto-plastic (Mohr–Coulomb) solid, 3D analysis 542–8
- elasto-plastic rate integration 270–275
- elasto-plastic solids (3D), forced vibration analysis 585–9
- element assembly technique 276
- element-by-element techniques 68–72
- element conductivity matrix 333, 344–6, 358
- element local coordinate systems 140f
- element-mass matrix 40, 128
- element matrix assembly, structure chart 78f
- element node numbering, shape functions and 611–17
- element stiffness
- derivation 32–35
  - integration and assembly 175
  - element stiffness matrix 47, 135, 155, 339
  - element strain energy 39
  - elements
    - 1D elements 611
    - 2D elements 35–8, 84–6, 612–14
    - 3D elements 86–90, 615–17
    - 4-node tetrahedron 92f, 199
    - 8-node brick-shaped 43f
    - assembly of 90–95
    - cuboidal 169
    - multi-element assemblies 66–8
    - plate-bending 44–7
    - plate-bending elements 44–7
- `elmat` (subroutines) 467
- `emb_2d_bc` (geom library) 263, 635
- `emb_2d_geom` (geom library) 263, 635
- `emb_3d_bc` (geom library) 316, 636
- `emb_3d_geom` (geom library) 316, 636
- embanking process 289–94
- energy, elastic plane elements 38
- Ensight Gold format 18, 527, 539, 546, 626, 630
- equation solution subroutines 97t
- equilibrium equations 50
- solution of 95–6
- equivalent nodal loads 605–10
- Exascale 597
- `exc_nods` (main library) 305, 626
- excavations 294–305
- forces formulation 298f
  - vertical cuts 305
- exercises
- coupled problems 459
  - eigenvalue problems 480–482
  - forced vibrations 521–2
  - material non-linearity 326–30
  - static equilibrium of linear elastic solids 224–32
  - static equilibrium of structures 159–68
  - steady state flow 361–8
  - transient problems (uncoupled) 418–22
- failure criteria 238–40
- failure function, calculation 257
- failure surfaces, corners on 243–50
- `find_g` (parallel library) 641
- `find_g3` (parallel library) 641

- find\_g4** (parallel library) 641  
**find\_pe\_procs** (parallel library) 641  
finite element analysis, parallel processing 523–602  
finite element computations 59–113  
finite element mesh, 8-node quadrilaterals 10  
finite elements  
    element stiffness matrix 63  
    spatial discretisation 25–56  
first order time dependent problems,  
    solution of 99–103  
**fkdiag** (main library) 626  
flexural stiffness 128  
flow equations, simplified 51–4  
flow of fluids, Navier–Stokes equations 48–50  
fluid elements, stiffness/conductivity matrix 83  
fluid flow, mass matrix 88  
fluids, flow of 48–50  
**fmacat** (main library) 626  
**fmcoem** (geom library) 293, 636  
**fmdsig** (subroutine) 80  
**fmglem** (geom library) 293, 636  
**fmpplat** (main library) 626  
**fmrmat** (main library) 626  
forced vibration analysis  
    3D elastic solids 581–5  
    3D elasto-plastic solids 585–9  
    elastic beams 483–8  
    elastic–plastic solids 512–17  
forced vibration analysis of elastic solids  
    using 4- or 8-node rectangular quadrilaterals 489–93  
    using 4-node rectangular quadrilaterals 504–8  
    using 8-node rectangular quadrilaterals (theta method) 493–7, 508–12  
    using 8-node rectangular quadrilaterals (Wilson’s method) 498–501  
forced vibrations 483–517  
    exercises 521–2  
    glossary 517–21  
    programs 483–517  
**form\_s** (main library) 628  
**forma\_a** (main library) 627  
**formke** (main library) 627  
**formku** (main library) 464, 627  
**formlump** (main library) 627  
**formm** (main library) 627  
**formnf** (geom library) 636  
**formtb** (main library) 627  
**formupv** (main library) 627  
**formupvw** (parallel library) 641  
FORTRAN 6  
    arithmetic 6–7  
    array features 9–16  
    conditions 7–8  
    intrinsic procedures 11–12  
    library routines 13–16  
    loops 8–9  
    typical simple program 5  
forward Euler method 272–4  
foundation stiffness 128, 153  
foundation stiffness matrix 128  
free surface flow, boundary conditions 346f  
**fsparkv** (main library) 628  
**g\_t\_g\_ns** (parallel library) 642  
**g\_t\_g** (parallel library) 642  
**gather** (parallel library) 641  
**gauss\_band** (main library) 628, 631  
general 2-(plane) or 3D analysis  
    consolidation equation 397–401  
    steady seepage 398  
    steady seepage (no global conductivity matrix assembly) 397  
**geom\_freesurf** (geom library) 346, 350, 636  
**geom** library subroutines 635–637  
**geom\_rect** (geom library) 246, 339, 409, 426, 442, 637  
geometric non-linearity 233  
**geometry\_8bxz** (parallel library) 641  
**geometry\_20bxz** (parallel library) 641  
geometry subroutines 169  
    hexahedron–xz 169  
**glob\_to\_axial** (main library) 628  
**glob\_to\_loc** (main library) 628  
global conductivity matrix 333, 339, 358  
global gravity loading vector 198  
global node numbering system 117  
global stiffness matrix 117–19, 128, 133, 143, 155

- global variables 526  
 GMRES (Generalised minimum residual) 70  
 GPU (graphics processing unit) 4, 17, 524, 589–94, 597  
 gravity loads 263, 294  
 hardware 2  
 heat conduction equation 51  
 hexahedron  
   8-node 615  
   14-node 89f, 202f, 616  
   20-node 192f, 617  
 hexahedron\_xz (geom library) 192, 637  
 global node and element numbering 194f  
 hinge (main library) 628  
 IF . . . THEN . . . ELSE structure 7–8, 16  
 image-based modelling 560–561  
 inconsistent tangent matrix 275  
 initial stress methods 242–3  
   stress redistribution 267  
 integration  
   for quadrilaterals 61–4  
   for triangles 65  
 interp (main library) 629  
 invar (main library) 246, 628  
 invert (main library) 76, 629  
 iterations, constant stiffness 233  
 Jacobi algorithm 96–98  
 laminar fluid flow (3D), element  
   conductivity matrix 88  
 lancz1 (main library) 77, 646  
 lancz2 (main library) 77, 647  
 Lanczos method 71, 77, 469–74, 455, 459  
 Laplace equation, solution of 333, 346  
 Laplacian flow (3D) 548–53  
 limiting bending moment 145  
 limiting torsional moment 145  
 linear elastic solids, static equilibrium of 169–232  
 linear strain rectangle 38  
 linmul\_sky (main library) 340, 629  
 load\_function (main library) 435, 629  
 load control analysis 143  
 load displacement behaviour 148f  
 loading (parallel library) 642  
 loc\_to\_glob (main library) 629  
 local coordinates  
   quadrilateral elements 59–64  
   triangular elements 64–6  
 loops 8–9  
 lumped mass approximations 99  
 main Library subprograms 623–33  
 make\_ggl (parallel library) 642  
 Mandel–Cryer effect 459  
 Maple (computer algebra system) 63  
 masking argument 12  
 mass approximations 33, 84, 99  
 mass matrix formation 83–4  
 material non-linearity 233–330  
   exercises 327–330  
   glossary 322–7  
   programs 244–70  
 MATMUL 11, 17, 69, 77, 80–83  
 Matrix  
   consistent tangent 275–6  
   inconsistent tangent 275  
 matrix displacement method 115  
 matrix–vector multiplication subroutine 77  
 mcdp1 (main library) 629  
 memory management 2–3  
 mesh (main library) 629  
 mesh (3D), numbering system and data 93f, 96f  
 mesh generation routines 92  
 mesh numbering 66–7, 67f  
 mesh partitioning 534–5  
 mesh\_size (geom library) 175  
 meshes  
   deformed 18–19, 21–3, 170, 216, 249–50, 264f, 318f, 541f  
   numbering system and data 66–7  
 message passing 4–5  
 method of fragments 341  
 METIS (mesh partitioning tool) 534–6, 538  
 minimum residual method (MIN-RES) 71

- mocouf (main library) 257, 630  
Mohr–Coulomb criterion 236, 238–9, 546  
Mohr–Coulomb failure function 239, 257, 311, 424, 546, 630  
Mohr–Coulomb (Tresca) surface 243  
moment–curvature relationship 34, 143  
  elastic-perfectly plastic 143, 143f  
MPI (message passing interface) 5  
  libraries 18  
  library routines 18, 523–4, 526
- Navier–Stokes  
  3D problems 94, 641  
  3D steady state analysis 429, 565  
  equations 48–50, 423
- Newmark method 109–10  
Newmark time-stepping 483–8  
nodal loads, equivalent 129–30, 135–6, 605–10  
node freedom arrays 94–5, 193, 303, 532  
non-axisymmetric analysis, axisymmetric  
  elastic solid 184–91
- non-vertical elements, transformation  
  angle 139, 139f
- norm (main library) 630
- ns\_cube\_bc20 (parallel library) 642  
ns\_loading (parallel library) 642  
num\_to\_g (main library) 630
- OpenHMPP 589  
OpenMP 4–5, 523, 597  
Optimised maths library 213, 220
- paging 2, 68  
ParaFEM 527, 534–5, 539, 639  
parallel and serial programs 524–36  
  pp appendage 527  
  differences between 524–5  
  domain composition 533–4  
  gathering and scattering 533  
  global variables 526  
  load balancing 535–6  
  MPI library routines 526  
  parallel libraries 525–6  
  reading and writing 530–532  
  reindexing 533  
  rest instead of nf 532
- parallel libraries 525–6  
parallel library subroutines 639–44  
parallel processing  
  benefits 524  
  effect of mesh subdivision 524
- parallel processing of finite element  
  analyses 523–602  
  glossary 597–602  
  programs 536–94
- parallel processors 4–5  
ParaView 18–20, 170, 212, 316, 527, 539–40, 546, 552, 575f, 579–80
- partial differential equations,  
  semi-discretisation 33t
- Pascal pyramid of polynomials 88–90  
Pascal’s triangle 89 89f
- pcg *see* preconditioned conjugate gradient  
  (pcg) technique
- penalty technique 128, 176, 268, 427
- pin\_jointed (main library) 630  
pin-jointed frames in 2D, 123  
pipelines 3  
planar elements 605, 606  
plane analysis of the  
  diffusion–convection equation 401–5  
  untransformed solution 405–10
- plane or axisymmetric analysis of the  
  consolidation equation ( $\theta = 0$   
  method) 390–395
- of the consolidation equation ( $\theta$   
  method) 397–401
- of steady seepage 337–44
- plane or axisymmetric consolidation  
  analysis 387–90
- plane or axisymmetric consolidation  
  analysis ( $\theta$  method) 380–386
- plane or axisymmetric strain analysis,  
  elastic solid 169–232
- plane elastic analysis  
  quadrilateral elements 77–81  
  triangular elements 81–2
- plane element mass matrix 40
- plane free-surface flow, analysis  
  using 4-node quadrilaterals 344–51
- plane general quadrilateral elements 60f
- plane problems, degrees of freedom per  
  node 82

- plane rectangular elements 60f  
 plane steady laminar fluid flow 83  
 plane steady state Navier–Stokes equation  
     analysis using 8-node rectangular quadrilaterals 424–9  
     analysis using 8-node rectangular quadrilaterals (element-by-element solution) 429–33  
 plane strain 38–40  
 plane strain bearing capacity analysis of elastic–plastic materials 244–50  
     initial stress method 265–70  
     no global stiffness matrix assembly 250–254  
     viscoplastic strain method 244–50  
 plane strain consolidation analysis of Biot poro-elastic solids, incremental version 571–6  
 plane strain consolidation analysis of Biot poro-elastic–plastic, materials 448–54  
 plane strain construction  
     elastic–plastic embankment 290–299  
     elastic–plastic excavation 300–305  
 plane strain earth pressure analysis,  
     elastic–plastic material 265–70  
 plane strain slope stability analysis,  
     elastic–plastic material 260–265  
 plane stress 35–8  
 plane triangular grid 145  
 plastic moments 143  
 plastic potential derivatives 619–22  
 plastic stress–strain matrices 619–22  
 plate elements, node and freedom numbering 155  
 plate stiffness matrix 116  
 plates, analysis using 4-node rectangular plate elements 153–7  
 Poisson equation 51  
 Poisson’s ratio 36, 41, 44, 47, 155, 158, 175–6, 193, 198, 209, 230, 258, 308, 325, 441, 456, 467, 516, 599, 625–6, 630, 632  
 pore pressure 308  
 potential 333  
 potential surfaces, corners on 243–4  
 preconditioned conjugate gradient (pcg) technique 100, 112, 211–13, 525, 539  
 preconditioned matrix 447  
 preconditioning 69–70  
 pressure shape functions 103  
 principal stress space 235–7, 237f, 239–40  
 principle of minimum potential energy 38  
 programming  
     finite element computations 59–113  
     using building blocks 75–95  
 programs  
     1D analysis of axially loaded elastic rods 116–21  
     1D analysis of steady seepage 334–7  
     1D consolidation analysis 370–373  
     2-(plane) or 3D analysis of steady seepage 351–5  
     3D analysis of Biot poro-elastic solid 571–576  
     3D analysis of elastic solid 191–6, 536–41  
     3D analysis of elasto-plastic (Mohr–Coulomb) solid 542–8  
     3D Laplacian flow 548–53  
     3D steady state Navier–Stokes analysis 565–71  
     3D strain analysis of elastic–plastic slope 313–18  
     3D strain analysis of elastic–plastic slope (no global stiffness matrix assembly) 209–13  
     3D strain of elastic solid 191–6  
     3D transient flow–explicit analysis in time 562–5  
     3D transient flow–implicit analysis in time 553–61  
 analysis of elastic beams 127–33  
 analysis of elastic pin-jointed frames 121–6  
 analysis of elastic rigid-jointed frames 133–41  
 analysis of elastic–plastic beams or rigid-jointed frames 141–9  
 analysis of plane free-surface flow 344–51

- analysis of the plane steady state  
Navier–Stokes equation 424–9
- analysis of the plane steady state  
Navier–Stokes equation (no global matrix assembly) 429–33
- analysis of plates 153–7
- axisymmetric “undrained” strain of elastic–plastic solids 308–13
- eigenvalue analysis of 3D elastic solids 576–81
- eigenvalue analysis of elastic beams 462–6
- eigenvalue analysis of elastic solid in plane strain 465–9
- eigenvalue analysis of elastic solid in plane strain (consistent mass) 469–74
- eigenvalue analysis of elastic solid in plane strain (lumped mass) 462–5
- forced vibration analysis of 3D elastic solid 581–5
- forced vibration analysis of 3D elasto-plastic solid 585–9
- forced vibration analysis of elastic beams 483–8
- forced vibration analysis of elastic solid in plane strain 489–93
- forced vibration analysis of elastic solid in plane strain (theta method) 493–7, 508–12
- forced vibration analysis of elastic solid in plane strain using rectangular uniform size 4-node quadrilaterals 489–93
- forced vibration analysis of elastic solid in plane strain (Wilson’s method) 498–501
- forced vibration analysis of elastic–plastic (von Mises) solid in plane strain 512–17
- forced vibrations 483–517
- general 2-(plane) or 3D analysis of the consolidation equation 397–401
- general 2-(plane) or 3D analysis of steady seepage 351–5
- general 2-(plane strain) or 3D analysis of elastic solids 195–204
- non-axisymmetric analysis of axisymmetric elastic solids 184–90
- plane analysis of the diffusion–convection equation (self-adjoint transformation) 401–5
- plane analysis of the diffusion–convection equation (untransformed solution) 405–10
- plane or axisymmetric analysis of the consolidation equation (theta method) 375–8
- plane or axisymmetric analysis of steady seepage 337–44
- plane or axisymmetric consolidation analysis 386–90
- plane or axisymmetric consolidation analysis (theta method) 380–386
- plane or axisymmetric strain analysis of an elastic solid 170–184
- plane strain bearing capacity analysis of elastic–plastic material 244–70
- plane strain bearing capacity analysis of elastic–plastic material (initial stress method) 265–70
- plane strain bearing capacity analysis of elastic–plastic material (no global stiffness matrix assembly) 250–254
- plane strain bearing capacity analysis of elastic–plastic material (viscoplastic method) 244–50
- plane strain consolidation analysis of Biot poro-elastic solid (incremental version) 438–44
- plane strain consolidation analysis of Biot poro-elastic–plastic material 448–51
- plane strain construction of elastic–plastic embankment 290–299
- plane strain construction of elastic–plastic excavation 300–305
- plane strain earth pressure analysis 265–70

- programs (*continued*)
- plane strain slope stability analysis 260–265
  - stability analysis of elastic beams 150–153
  - static equilibrium of linear elastic solids 169–232
  - steady state flow 333–69
  - timings of vectorised 221t
  - propagation problems 48, 68, 72
- quadrilateral elements
- global matrix assembly for mesh 67t
  - local coordinates 59–64
  - mesh 66f
  - plane elastic analysis 77–81
- quadrilaterals
- 4-node 177, 179f, 606, 609, 614
  - analytical integration 63–4
  - numerical integration 61–3
- quadrilaterals (8-node) 84–5, 179, 512, 606
- element stiffness matrix 182
  - global node and element numbering 179f
  - local node and freedom numbering 181f, 189f
- quadrilaterals (9-node) 606, 614
- local node and freedom numbering 199f
- ramp loading 422f, 444f, 575
- random field 594–5
- Rankine passive mechanism 270
- Rayleigh damping coefficients 107
- `rearrange` (parallel library) 643
- `rearrange_2` (parallel library) 643
- `rect_km` (main library) 630
- rectangular plate bending elements 45f
- rectangular quadrilaterals, plane or axisymmetric consolidation analysis 380–386
- `rigid_jointed` (main library) 631
- `rod_km` (main library) 631
- `rod_mm` (main library) 631
- rod elements 25–28
- 2-node rods 611
  - node and freedom numbering 120f
- node and freedom numbering (2D) 124f
- node and freedom numbering (3D) 126f
- rod mass element 28
- rod stiffness matrix 25–7
- routines, special purpose 76
- `sample` (main library) 175, 631
- scalar characters, glossary 158, 222, 325, 360, 417, 456, 478–9, 519, 599
- scalar computers 3
- scalar integers, glossary 157–8, 221, 322–3, 359–360, 416, 454–5, 477–8, 517–18, 597–8
- scalar logicals, glossary 158, 222, 325, 360, 417, 456, 479, 519, 600
- scalar potential problems, degrees of freedom per node 94
- scalar reals, glossary 158, 222, 323–5, 360, 416–17, 455–6, 478, 518–19, 598–9
- `scatter` (parallel library) 525, 643
- scattering, gathering and 533
- second order time dependent problems, solution of 106–13
- `seep4` (main library) 631
- seepage analysis 341, 398
- SELECT CASE, construct 8, 15
- selective reduced integration (SRI) 230
- serial and parallel programs, differences between 524–36, 575, 579
- `shape_der` (main library) 631
- `shape_fun` (main library) 631
- shape functions, and element node numbering 611–17
- shared memory systems 5
- software, applications 5–9
- soil stiffness 132
- solid elements, testing admissibility 202
- solids, element equations summary 47–8
- solution of coupled transient problems 104–6
- absolute load version 105–6
  - incremental load version 106, 438, 445
- solution of first order time dependent problems 99–103

- solution of second order time dependent problems 106–13  
damping 108–9  
explicit methods 113  
inclusion of forcing terms 109  
modal superposition 107–9  
Newmark or Crank–Nicolson method 109–10  
Wilson’s method 110–11
- `solve_band` (main library) 628, 631
- `spabac_gauss` (main library) 435, 464, 632
- `spabac` (main library) 176, 246, 340, 371, 496, 631
- `sparin_gauss` (main library) 435, 442, 464, 632
- `sparin` (main library) 176, 246, 340, 371, 496, 507, 632
- spatial discretisation, by finite elements 25–56
- stability** (main library) 632
- stability analysis, of elastic beams 150–153
- static equilibrium of linear elastic solids 169–224  
exercises 224–32  
glossary 221–4  
programs 169–216
- static equilibrium problems 68
- static equilibrium of structures 115–68  
exercises 159–68  
glossary 157–9  
programs 115–57
- static problems 47
- steady seepage 1D analysis 334–7  
general 2-(plane) or 3D analysis 351–5  
general 2-(plane) or 3D analysis (no global conductivity matrix assembly) 355–9  
plane or axisymmetric analysis 337–44  
under sheet pile wall 338, 341, 364f
- steady state equations 49
- steady state flow 333–59  
exercises 361–6  
glossary 348–50  
programs 334–59
- steady state Navier–Stokes analysis (3D) 565–71
- steady state problems 423
- stiff spring technique 128, 176, 371
- `stiff4` (main library) 632
- storage-saving strategies 68, 95, 96, 97f, 112–13
- storage strategies 90f
- strain (3D), elastic solid 90, 581–5
- strain analysis (3D) of an elastic–plastic slope 313–18
- strain analysis (3D) of elastic–plastic slope, viscoplastic strain method 319–22
- stress, and strain (3D) 42–4
- stress invariants 236–8
- stress redistribution, initial stress method 242–3, 265–70
- stress–strain behaviour 235–6
- structure charts 15, 100, 101–2, 144f, 172f, 213f, 247f, 280, 410, 428f, 443f, 464f, 492f
- Biot analysis 443f
- element matrix assembly 23, 78f
- element-by-element product algorithm 102f, 393–7
- explicit time integration 377–80, 390–393, 512–17
- implicit analysis of transient problems 369
- matrix multiplication 16f
- Navier–Stokes analysis with global matrix assembly 428f
- pcg algorithm 213f
- tangent stiffness approach 275–89, 280f
- structured programming 15–17
- structures, static equilibrium 115–68
- subdomains 533–4, 536
- subprogram libraries 13–15
- subroutines 95  
equation solution 97t  
geom library subroutines 635–7  
main library subroutines 623–33  
matrix–vector multiplication 72  
parallel library subroutines 639–44  
for solution of linear algebraic equations 76t

- subroutines (*continued*)  
     special purpose 90
- symmetric eigenvalue systems 71–2
- symmetric non-positive definite equations  
     71
- tangent stiffness methods 243, 275–85
- Terzaghi's 1D consolidation theory 53
- tetrahedral elements 90
- tetrahedron  
     4-node 92f, 199, 201f, 534, 615  
     constant strain 90  
     local node and freedom numbering  
         201f
- thermoelasticity 39–40, 169
- transient analysis, mesh-free strategies  
     380–386
- transient conditions 53
- transient flow (3D)  
     explicit analysis in time 562–5  
     implicit analysis in time 553–61
- transient problems 56, 423
- transient problems, first order (uncoupled)  
     369–422  
     exercises 418–22  
     glossary 416–18  
     programs 369–416  
     structure chart for implicit analysis 369
- TRANSPOSE, FORTRAN intrinsic  
     function 80
- Tresca failure criterion 236, 306
- Tresca surface 243
- triangles  
     3-noded, 170–184, 174f, 198  
     6-noded, 170–184, 198, 81, 577, 584  
     10-noded, 170–184, 198, 577, 579,  
         584  
     15-noded, 170–184, 177f, 198
- node numbering system 176–7
- numerical integration 65
- triangular elements  
     local coordinates for 64–6  
     plane elastic analysis 81–2
- two-bay portal frame, proportional  
     loading 145
- uncoupled problems 56, 105, 369–422,  
     435
- exercises 418–22
- programs 369–422
- structure chart for implicit analysis  
     369
- undrained soil analysis 305–22
- unsymmetric systems 70–71
- variable names, glossary 221–4, 322–7,  
     359–61, 416–18, 454–8, 477–80,  
     517–21, 597–602
- variable (tangent) stiffness method 233,  
     234f, 243, 275–6
- vecmsh (main library) 633
- vector processors 3–4
- vector subscripts 10–11
- velocity shape functions 103
- vertical elements, transformation angle  
     139, 140f
- viscoplastic algorithm, structure chart  
     247f
- viscoplasticity 235, 240–241
- vmdp1 (main library) 633t
- vmflow (main library) 633t
- vmp1 (parallel library) 644t
- von Mises criterion 238–239, 268
- water, flow through dams 348
- Wilson's method 110–111