FELIPE User Manual [1]

M. B. Reed
Department of Mathematical Sciences
University of Bath
Bath BA2 7AY, U.K.
email: Martin@felipe.co.uk
fax: +44 (0)1225 386492

[1]Last modified: July 2005

Acknowledgements

I am very grateful to Prof. J. R. Whiteman and Dr. M. K. Warby for permission to use in FELIPE some of the basic graphics and PostScript subroutines developed by Dr. Warby, and to Dr. T.-Y. Chao for working with me on the programming and documentation of the elasticity module. I also acknowledge the support of the Enterprise in Higher Education Unit at Brunel University, in enabling me to work on this project. The Fortran coding of the elasticity 'main engines' is based on the FINEPACK program developed at the Dept. of Civil Engineering, University College Swansea, and I am grateful to Dr. D. J. Naylor for permission to use this.

Contents

Chapter 1

Introduction

FELIPE (Finite Element Learning Package) is a software package whose primary objective is to help students understand the finite element method in mathematics and engineering, and develop their own f.e. programs. Its advantage over the f.e. textbooks which provide their example programs printed or on ftp or cd-rom, is that it combines full, commented and documented source code (in standard Fortran77) for the f.e. 'main engines', with powerful interactive graphics pre- and post-processors capable of generating complex, detailed meshes. Because of this, it is also very suitable for practising engineers and researchers as a low-cost alternative to the many commercial "black box" packages on the market (which do not provide source code).

The principal components of FELIPE are:

- PREFEL, a pre-processor, used to create the input data file which defines the finite element mesh, boundary conditions, material properties, loading, etc. The resulting data file is given a `.dat` filename extension. The pre-processor is provided as the executable file `PREFEL.EXE`;

- Three Basic-level Fortran77 'main engines' (for the 2D Poisson's equation, plane elasticity, and beam/frame analyses), the theory for which is summarized in this Manual. Each 'main engine' reads an input file such as `prob1.dat` (created by the PREFEL pre-processor) and produces an output file `prob1.out` (for use with the post-processor) and a results file `prob1.prt` in printable format;

- Six further Advanced-level Fortran77 'main engines' for analysing a range of mathematics and engineering applications (e.g. viscoplasticity, thermoelasticity), which illustrate the main practical aspects of finite element programming. In particular, a wide range of 1D and 2D finite and infinite element types are used, and coding is provided for all the most important algorithms for equation solution (from Gaussian elimination to conjugate gradients with Incomplete Choleski preconditioning). The individual 'main engines' are summarized below. There is also a file

of input/output subroutines common to all the 'main engines'. Each 'engine' is provided in source code `.for` and executable `.exe` form;

- linking files (`.inf`) for each of the 'main engines' if they are to be compiled, linked and run using the Salford FTN77 compiler;
- FELVUE, a post-processor which reads in a `.out` file and displays the results graphically. Displays include contouring, stress crosses, displacement vectors, deformed mesh, etc. This processor is also provided in executable form, namely in the file `FELVUE.EXE`. It is possible to produce PostScript files (with a `.ps` extension) of the graphical displays, for later printing;
- `SALFLIBC.DLL`. The Salford Fortran library needed to allow the programs to run on any PC;
- Sample `.dat` and `.out` files, for one or more example problems for each 'main engine', also documented in this Manual and shown on the FELIPE website.

By this means, users of FELIPE have the interactive graphics processing power of a commercial finite element package, combined with fully-documented source code which they can modify and extend for their own purposes. The package is completely self-standing; the only supporting software needed is a Fortran compiler if the user wishes to modify the source code and re-compile it. Since the pre- and post-processors communicate with the main f.e. programs through formatted ASCII data files, they can also be interfaced with other finite element programs, whether in Fortran or another language.

The pre- and post-processors are compiled under FTN77 (as are the executable versions of the 'main engines'), which includes a memory extender; thus, large meshes can thus be created, and problems of real mathematical and engineering significance solved. (The processors are dimensioned to handle a maximum of 900 elements, and 3,000 nodes.) They make extensive use of the graphics and mouse routines available with the FTN77 compiler, and can also produce PostScript graphics files for subsequent processing by, for example, GhostScript software (available from www.hensa.ac.uk by ftp) and printing on LaserJet or DeskJet printers.

The 'main engines' can also be compiled and run using FTN77 (which is available for personal use free from the Salford software website), but if this is not available any other Fortran compiler can be used, as they are written in standard Fortran77.

1.1 The nine 'main engines'

The three Basic-level 'main engines', and their principal features, are now listed:

1. `POISS.FOR`
   Application: solves Poisson's equation $-a_x u_{xx} - a_y u_{yy} = f(x, y)$ on an arbitrary 2D domain.
   Material properties: diffusion coefficients a_x, a_y
   Primary nodal unknown: potentials U
   Secondary unknowns: flow rates U_x, U_y
   Elements used: 3-noded linear triangles.
   Boundary conditions: reflecting, radiating and Dirichlet boundaries
   File size: 25.8KB
   Analysis type: linear
   Matrix storage: symmetric band
   Solution algorithm: Choleski ($L.L^T$) decomposition

2. `ELAST.FOR`
   Application: solves plane strain or plane stress linear elasticity problems
   Material properties: Young's modulus E, Poisson's ratio ν, thickness t, tensile strength σ_{ten}
   Primary nodal unknown: displacements u, v
   Secondary unknowns: stresses $\sigma_x, \sigma_y, \tau_{xy}$
   Elements used: 8-noded 'serendipity' quadrilaterals
   Boundary conditions: fixities in x or y planes
   Loading: point loads, surface tractions
   File size: 34.8KB
   Analysis type: linear
   Matrix storage: symmetric band
   Solution algorithm: Choleski ($L.L^T$) decomposition

3. `FRAME.FOR`
   Application: analyses plane frames comprising elastic beams.
   Material properties: Young's modulus E, Moment of Inertia I, cross-sectional area A
   Primary nodal unknown: displacements x, y and rotations θ
   Elements used: 2-noded cubic beam elements.
   Boundary conditions: displacement and rotation fixities
   Loading: point loads, surface tractions
   File size: 24.4KB
   Analysis type: linear
   Matrix storage: element-by-element matrices on scratch file
   Solution algorithm: preconditioned conjugate gradients, with diagonal preconditioning

The six Advanced-level 'main engines' are:

1. `ELADV.FOR`
   Application: Large-scale elasticity analyses
   Material properties: Young's modulus E, Poisson's ratio ν, thickness t, tensile strength σ_{ten}
   Primary nodal unknown: displacements u, v
   Secondary unknowns: stresses $\sigma_x, \sigma_y, \tau_{xy}$
   Elements used: 3- and 6-noded triangles, 4- and 8-noded quadrilaterals, mapped infinite elements, 2- and 3- noded (cubic and quartic) beam elements
   Boundary conditions: fixities in x or y planes
   Loading: point loads, specified displacements, surface tractions, body forces, excavation loading
   File size: 91.9KB
   Analysis type: linear
   Solution algorithm: symmetric frontal algorithm

2. `PLAST.FOR`
   Application: plane strain associated-flow Mohr-Coulomb elasto-plasticity analyses
   Material properties: Young's modulus E, Poisson's ratio ν, triaxial stress factor k, strength σ_c
   Primary nodal unknown: displacements u, v
   Secondary unknowns: stresses $\sigma_x, \sigma_y, \tau_{xy}$
   Elements used: 8-noded 'serendipity' quadrilaterals
   Boundary conditions: fixities in x or y planes
   Loading: point loads, surface tractions
   File size: 50.3KB
   Analysis type: nonlinear, iterative, incremental
   Matrix storage: symmetric band
   Solution algorithm: Choleski ($L.L^T$) decomposition

3. `VPLAS.FOR`
   Application: plane strain Mohr-Coulomb elasto-viscoplasticity analyses, with non-associated flow
   Material properties: Young's modulus E, Poisson's ratio ν, triaxial stress factor k, strength σ_c, fluidity γ, dilation factor l
   Primary nodal unknown: displacements u, v
   Secondary unknowns: stresses $\sigma_x, \sigma_y, \tau_{xy}$
   Elements used: 8-noded 'serendipity' quadrilaterals
   Boundary conditions: fixities in x or y planes
   Loading: point loads, surface tractions
   File size: 75.5KB
   Analysis type: nonlinear, incremental, time-dependent
   Solution algorithm: Frontal algorithm, for non-symmetric matrices

4. `PLADV.FOR`
   Application: as `PLAST`, but with a range of solution algorithms
   Material properties: Young's modulus E, Poisson's ratio ν, triaxial stress factor k, strength σ_c
   Primary nodal unknown: displacements u, v
   Secondary unknowns: stresses $\sigma_x, \sigma_y, \tau_{xy}$
   Elements used: 8-noded 'serendipity' quadrilaterals
   Boundary conditions: fixities in x or y planes
   Loading: point loads, surface tractions
   File size: 67.7KB
   Analysis type: nonlinear, iterative, incremental
   Matrix storage: symmetric skyline, element-by-element
   Solution algorithms: Choleski ($L.L^T$) and $L.D.L^T$ decomposition, conjugate gradients with diagonal or Incomplete Choleski preconditioning

5. `THERM.FOR`
   Application: plane stress/strain thermoelasticity
   Material properties: Young's modulus E, Poisson's ratio ν, thickness t, conductivity coefficient k, coefficient of thermal expansion α
   Primary nodal unknown: displacements u, v, temperatures T
   Secondary unknowns: stresses $\sigma_x, \sigma_y, \tau_{xy}$
   Elements used: 4-noded (linear) and 8-noded (serendipity) quadrilaterals with (u, v, T) degrees of freedom at all nodes
   Boundary conditions: fixities in x or y planes, reflecting and Dirichlet temperature boundaries
   Loading: point loads, surface tractions
   File size: 44.2KB
   Analysis type: linear, coupled
   Matrix storage: nonsymmetric band
   Solution algorithm: Gauss elimination for non-symmetric matrices

6. `CONSL.FOR`
   Application: plane strain soil consolidation (poroelasticity)
   Material properties: Young's modulus E, Poisson's ratio ν, effective permeabilities $\frac{k_x}{\gamma_w}, \frac{k_y}{\gamma_w}$, effective porosity $\frac{\eta}{K_f}$
   Primary nodal unknown: displacements u, v, pore-pressures p
   Secondary unknowns: effective stresses $\sigma_x, \sigma_y, \tau_{xy}$
   Elements used: 8-noded 'serendipity' quadrilaterals with pore-pressure d.o.f.s at corner nodes only
   Boundary conditions: fixities in x or y planes, impermeable or permeable boundaries
   Loading: point loads, surface tractions

File size: 53.4KB
Analysis type: linear, coupled, time-dependent
Matrix storage: symmetric band
Solution algorithm: $L.D.L^T$ decomposition

1.2 Installation

To install the package from the floppy disk, run the self-extracting zip file `FULLDISK.EXE` which is on the disk. You can do this by locating the file on your disk drive (normally the A: drive) using *My Computer* or *Windows Explorer*, and double-clicking on it. Alternatively, type `a:\fulldisk.exe` into the command line copy using the *Run...* utility from the Start menu. You will be prompted to nominate a directory into which the FELIPE files should be unzipped; the default is `C:\FELIPE`.

If you have already downloaded the evaluation version of FELIPE from the website into the `C:\FELIPE` directory, you can still use the same directory; the demonstration versions of the files will be overwritten by the full versions, and new files added.

Because of size limitations, not all of the example datafiles provided with the demonstration version of FELIPE are included on the disk; they can be obtained by downloading the zipped files DATAFILES.ZIP or DATAFILES.EXE from the FELIPE website.

The installation process does not alter any Windows settings on your PC. To uninstall FELIPE, simply delete the directory containing the files.

In this manual, Chapter 2 describes how to use the PREFEL pre-processor.

The next three chapters describe the three Basic-level 'main engines': Chapter 3 covers the theory and programming of the Poisson solver, while Chapter 4 describes the solver for elasticity problems, and Chapter 5 deals with beam theory.

Chapter 6 describes the use of the FELVUE post-processor. Then

Chapter 7 summarizes the operation and use of the other six, Advanced-level 'main engines'. Chapter 8 covers the various algorithms used for equation solution.

Chapter 9 documents the sample datafiles provided in the FELIPE package.

The final Chapter suggests ways in which the 'main engines' may be modified, and new 'main engines' written, and gives recommendations for textbooks for further reading about the finite element method.

Chapter 2

Using the pre-processor

2.1 Preliminaries

Before starting to use PREFEL, you must define the problem, and draw up a coarse finite element mesh. The first thing to do, is to decide which type of problem you are interested in using FELIPE to solve. Is your problem one in which the primary unknown quantity is a vector with x and y components (for example a field of displacements or flow velocities), or one in which the primary unknown is a scalar variable (such as temperature or pressure)? FELIPE uses the elasticity 'main engine' ELAST.FOR as an example of the 'vector' class of problems, and the 'main engine' POISS.FOR (solving Poisson's equation over a 2D region) to illustrate the 'scalar' class of problems. In the analysis of beams, there are three primary unknowns: the x and y displacements and the rotation at each node.

New users of FELIPE should first practise on solving Poisson, elasticity and beam problems at Basic level, before moving to develop their own 'main engine' and produce data files for it at Advanced level. A brief description of each standard problem now follows:

- **Elasticity**. In elasticity problems, the material properties which must be defined for each element, are the Young's modulus E, the Poisson's ratio ν and the material thickness t. Loading consists of point loads applied at nodes, and surface tractions applied on element boundary edges. The mesh represents a linear elastic material in either plane stress (zero out-of-plane stress) or plane strain (*i.e.* no movement in the out-of-plane direction).

- **Poisson**. Here, we seek to solve numerically the Poisson equation (a.k.a. the quasi-harmonic equation):

$$-a_x \frac{\partial^2 u}{\partial x^2} - a_y \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{2.1}$$

over a 2D region, with appropriate boundary conditions; these may be:

- Dirichlet boundary: fixed values of U on the boundary;
- reflective boundary: zero gradient or flow across the boundary;
- radiating boundary: flow fixed or proportional to the temperature.

The parameters a_x, a_y must be defined for each element, in the same way as the material parameters E, ν in elasticity problems. The right-hand-side function $f(x, y)$ will be defined in a FUNCTION routine in the program `POISS.FOR`.

- **Beam**. The axial stiffness of the beam is defined by its Young's modulus E, and its cross-sectional area A. Its flexural stiffness is defined by the Moment of Inertia I. Apart from the material properties, the preparation of a mesh for a beam or frame problem is a simplified version of the procedure for 2D elasticity problems, and will not be dealt with separately.

The next step is to draw up a coarse mesh. If you will be using the programs `ELAST.FOR` or `POISS.FOR` , then the mesh must consist of 8-noded quadrilateral elements (for elasticity problems) or of 3-noded linear triangles (for Poisson problems) respectively. In the latter case, however, it is possible to create a mesh of 4-noded linear quadrilaterals, which can be refined (see §2.4.3), and finally triangulated (cutting each quadrilateral into two triangles). For `BEAM.FOR`, each member of the frame is represented by a two-noded beam element. The range of elements available is illustrated in the Appendix.

Determine the coordinates of all nodes in your mesh.

Note that if using 8-noded quadrilaterals, you can create elements with curved sides, when the three nodes making up the side are not colinear. The curve is determined by the coordinates of the three nodes, and the quadratic basis functions in one dimension.

Number all the elements, and all the nodes; note that the numbering will be changed later, in the pre-processor.

Values of the parameters E, ν, t (for Elasticity problems), a_x and a_y (for Poisson problems) must be associated with each element. These values will henceforth be referred to as the material properties of the element.

The mesh boundary conditions must also be specified — that is, the parts of the boundary which are fixed in one or both directions (for elasticity) or which have Dirichlet boundary conditions (for Poisson problems) must be specified — in the main mesh input section.

The above data defining the mesh will be referred to as Part A of the full data required for input to a 'main engine. Part B data consists of the applied loads (for elasticity – see §2.6), any boundary inputs (e.g. specified values of U on Dirichlet boundary planes) and any incremental loading or timestepping regime data.

The recommended steps for preparing a full data file (which have been followed in the examples provided in the package), are:

1. produce a file containing the Part A data describing the coarse mesh, at Basic level, and save it with a filename such as test1.dat;
2. modify the Part A data by mesh refinement, element and node renumbering, etc. (if necessary changing to Advanced level), to produce the Part A data describing the final mesh, in a file named test2.dat;
3. add Part B data for the desired applied loading, saving the full data file as test3.dat, and run the appropriate 'main engine with this as the input file;
4. Use the pre-processor to add new Part B loads to test2.dat — or to edit the Part B loads in test3.dat — producing new datafiles test4.dat and so on, to see the effects of different load regimes.

It is however possible to proceed to add Part B data directly to the mesh input in test1.dat, if desired.

Note that whenever using the pre-processor to read in and edit an existing data file, the new file being produced must be given a different name. This is because PREFEL reads the input file and writes to the output file simultaneously.

As mentioned above, once the coarse mesh has been defined, it will be displayed graphically and the user can refine it in various ways. The element and node numbering of the resulting mesh can then be re-numbered to reduce the bandwidth of the associated global stiffness matrix.

2.2 Creating a data file

Run the pre-processor PREFEL by double-clicking on its icon in the `C:\FELIPE` directory.

A Text dialogue window will appear. This is used for text input. Note that in answering yes/no questions, it suffices for you to press the `y` or `n` keys; you do not need to press Return or Enter. The same applies in instances where only a single digit is

expected. Also note that if you press the Esc key in answer to a question, you will be able to exit the program without saving. It is worth pointing out that when running the compiled 'main engines, you do need to press Return in answer to questions. This is because the 'main engines use only standard Fortran77 which can be compiled by any compiler, and this does not include a "getkey routine for intercepting keyboard input.

Once some basic data has been entered in the Text window, a Graphics window will appear. Dialogue will then switch between the Text and Graphics windows; you will be prompted when to bring the appropriate window to the top of the desktop (by clicking on its caption bar). In the Graphics window the available options will appear as a menu on the right-hand side of the window. You choose a menu item by left- clicking on it; if you right-click on a menu item a **Help screen** *will appear.*

Next, you will be asked if you wish to create a new `.dat` file; answering `y` will invoke the CREATE subroutine, which will now be described. (By answering `n` you can modify an existing file, as described in the next section)

First, you give the name for the data file you will create (Do not type the `.dat` filename extension). The name should have a maximum of 8 characters (excluding the `.dat` extension); if longer, it will be truncated. A useful convention is to end it with a digit indicating if it is the 1st, 2nd, etc. mesh in the problem definition process, e.g. `myprob1.dat`. It must be a valid MS-DOS name - no spaces or non-alphanumeric characters.

By default, the datafile will be created in the same folder as that in which the pre-processor is stored. You will be asked if this is OK; if you answer 'n' you will be able to select a new folder from a Windows selection box.

Next, you can enter an identifying title for the mesh (max. 50 characters, including spaces).

Now you specify the type of problem to be solved, as described above:

- type `1` for a scalar-variable problem: at Basic level this will be a Poisson analysis;
- or type `2` for a vector-variable problem: an elasticity analysis at Basic level.
- or type `3` for a beam or frame problem.

(The simplest analysis to start with, is a Poisson problem). Note that the number you type (1, 2 or 3) indicates the number of basic unknowns at each node, used in creating the main graphical display in the post-processor.

You then specify whether you will be performing a Basic or Advanced level analysis. The 'main engines' `ELAST.FOR`, `POISS.FOR` and `FRAME.FOR` are written for Basic-level analyses only, so if you plan to use one of these 'main engines' you should reply `b`. The main additional features of Advanced-level analyses are that all the 1D and 2D

element types can be used in the mesh, and there is a wider range of material properties and loading régimes in elasticity analyses. A student exercise might be to modify the `ELAST.FOR` program to use the four-noded linear quadrilateral element; meshes with this element would be prepared using the Advanced option. Type `a` for Advanced or `b` for Basic – new users should use Basic level.

The type of analysis (scalar or vector variable), and its level (Basic or Advanced), is stored in an integer variable named `NDOFN` which denotes the number of degrees of freedom per node. In a vector-variable mesh there are two degrees of freedom per node (in elasticity the x- and y- displacements) while in a scalar-variable mesh there is only one degree of freedom per node: the potential U. In a frame, there are three degrees of freedom: the x- and y- displacements, plus the rotation of the beam at the node.

An Advanced-level mesh is denoted by setting NDOFN to -1 or -2 or -3 rather than 1 or 2 or 3. The value of NDOFN can be seen in the first line of data in the resulting `.dat` file. (At Advanced level, additional degrees of freedom can be defined at a later stage). At advanced level, you can also define additional parameters to use in your own customized 'main engines'.

You now have the option of defining the **basic mesh data** (nodal coordinates and element-node connections) either graphically using the mouse, or by keyboard entry.

If you have chosen the **keyboard option** for describing the mesh, enter the total number of nodes and the total number of elements; these integers are stored in `NPOIN` and `NELEM`. These are also written in the first data line of the `.dat` file — see §2.8. Then you enter the x and y coordinates of each node in turn (Type the two coordinates, separated by a space or comma, and press Return). Once all the nodal coordinates are entered, the screen will list them, and give you the chance to correct any errors. Then it will open a Graphics window, and plot the nodes in the x, y-plane so that you can check them. A pop-up box asks if you want to make any changes, *i.e.* to alter any of the coordinates. The focus reverts to the Text window, either for you to correct a nodal coordinate or to proceed to list, for each element in turn, the node numbers of the nodes around the element. To see the Text window you will need to click on its caption bar to bring it to the foreground. During the mesh-creation process, you will be prompted when you need to click on the Text or Graphics windows to make them visible.

It is important that the nodes around each element should be listed **anticlockwise**, starting at a corner of the element, as illustrated in the chart at the end of this chapter. A full list of the element types available is given in the Table below, but if you are performing a Basic-level analysis, the pre-processor will only permit you to use the corresponding element types: linear triangles or quadrilaterals (to be triangulated later) for Poisson problems (NDOFN=1), 8-noded quadrilaterals for elasticity (NDOFN=2), 2-noded line elements for beams (NDOFN=3). The element type is identified by specifying the number of sides (`NSIDE`) and the number of nodes (`NNODE`) in it. The full range of types available, with the corresponding `NSIDE` and `NNODE`, are given in the following table:

NSIDE	NNODE	Element type
1	2	2-noded 1D line element
1	3	3-noded 1D line element
2	6	Quadratic joint element
2	4	infinite joint element
3	3	Linear triangle
4	4	Linear quadrilateral
3	6	Quadratic (6-noded) triangle
4	6	Thin (6-noded) quadrilateral
4	8	Serendipity (8-noded) quadrilateral
3	4	Linear infinite element
3	5	Quadratic infinite element
2	3	Corner infinite element

For each element in turn, enter `NSIDE` and `NNODE` — as each of these will be a single digit, you do not need to press Return — followed by the node numbers around the element (separated by commas or spaces). You will be asked to enter a digit specifying the material type LMTYPE of the element. This is independent of the material properties, and can be used to indicate the type of behaviour (e.g. 1=elastic, 2=elasto-plastic). LMTYPE is used in the consolidation 'main engine' `CONSL.FOR`, where LMTYPE=1 for structural elements and LMTYPE=2 for soil elements (with pore pressure degrees of freedom). For Basic level analyses, just enter 1 for LMTYPE. When all elements have been described, the element-node connections will be listed. You can make changes to correct errors, and the data is also checked to make sure that all the nodes which were given coordinates, have been used in the mesh. If an element is to be deleted, the pre-processor will automatically delete any nodes which thus become unused, and re-number the remaining nodes.

As an alternative to the above, a simple mesh can be defined **graphically**. If you choose this option, you first enter the extent of the mesh in the x and y directions. A graphics screen with this region is then displayed, and you click on it to define the node positions. When the nodes have been defined, you define the elements. For each element you define in a popup window its type (by `NSIDE` and `NNODE`) and material type; then click on the nodes around the element anticlockwise, in the graphics screen. Note that you must start at a corner node and proceed anticlockwise, as shown in the chart at the end of this chapter. This method can be used for beam, finite and infinite elements, but not for joint elements. Curved sides can be defined. If you subsequently wish to make changes, this can be done as for the keyboard-entry method.

Once a consistent mesh has been accepted, it will be displayed in the Graphics window, and you can now examine the mesh by clicking the mouse on the menu bar at the right-hand-side of the display. Here, there are options to display the element numbers and node numbers on the mesh, or to pick a particular node or element by the

mouse or by typing in its number. There is also a zoom facility; click on 'Zoom' and then specify a rectangular window by clicking on two opposite corners of the window on the display. Note that if the zoom area is too small, and does not have any elements fully within it, a blank graph will result! An error message in this case will advise you to restore the full mesh and try a larger zoom area. When you are satisfied with the mesh, click on 'Continue' in the bottom right-hand-corner of the screen.

The next job is to specify the nodes at which **fixed or Dirichlet boundary conditions** apply. This is done graphically, by clicking the mouse on the appropriate nodes, for the conditions specified in the yellow pop-up box at the top of the screen. Note that for elasticity problems, there are three types of fixity which can be applied at a node:

1. fixed in the x-direction, but free to move in the y-direction;
2. free to move in the y-direction, but fixed in the x-direction;
3. fixed in both directions.

The three types of fixity are recorded by a two-digit fixity code: 10, 01 and 11 respectively. (A 1 in the i'th digit indicates that the displacement in the i'th coordinate direction is fixed). In a beam analysis (NDOFN=3) you will also be asked whether to fix the rotation; the displacement and rotation fixities are combined in a three-digit fixity code where the third digit refers to the rotation.

For Poisson problems, the Neumann or reflective boundary condition (no outward normal flow on the boundary) is the natural boundary condition for the problem, and parts of the boundary with this condition need no special treatment. Nodes with Dirichlet boundary conditions (specified values of the variable U) should be clicked in this section; they will be given a fixity code of 1. The actual value of U which is taken at the node, will be determined by defining values on Dirichlet planes in the "loading" section at the end; alternatively, a function $g(x, y)$ in the 'main engine' `POISS.FOR` may be used to fix these at run-time.

The whole mesh is displayed again, but this time the parts of the boundary between fixed nodes are displayed in grey or black — grey if fixed in one direction only, black if fixed in both. (Note that this does not strictly apply when there are more than two d.o.f.s per node!)

Next, the **material property sets** are entered, in the Text window. You may enter sets of properties, each corresponding to a different material; in the next data block you will associate these material property sets with the elements in the mesh. Note that set 1 will be the default material property set; any elements not associated with a specific set will be given the properties of set 1. If you have defined more than one property

set, you will, for each set in turn, be prompted to click on the elements associated with that set. (To choose an element, click the mouse on the centre-of-gravity of the element, which is highlighted by a white dot).

In Basic-level analyses, there are two, three or four components in each property set;

- Young's modulus E, Moment of Inertia I and cross-sectional area A for beams;
- the elastic parameters E and ν, and material thickness T for elasticity; you can also define a tensile strength σ_{ten} as component 4;
- and the coefficients a_x and a_y for Poisson's equation.

These will be stored as components 1,2 and 3 respectively. In Advanced-level analyses you are prompted for values of nine material property components; the additional components can be used for nonlinear analyses, for example in elasto-plasticity analyses the strength and coefficient of friction can be input (Note that the plasticity programs are for plane strain, and do not require input of material thickness — the plasticity parameters are input to property components 3 onwards.) Details of the material constants used in the Advanced-level programs supplied with FELIPE, are given in Chapter 7.

The number of property sets you describe, will be written as an integer parameter NPROS, along with NELEM (no. of elements), NPOIN (no. of nodes) and NDOFN (no. of degrees of freedom per node), as the basic mesh details in the first data line after the title.

Once this has been done, the basic data set is complete. A print file (with filename test1.prt if the data file is test1.dat) will also be written, echoing the mesh details in a readable format (read using Notepad or a word processor). You can now exit the pre-processor, saving this data file, or you can modify it by for example refining the mesh and adding loading, to produce a new data file. You can also choose to add loads directly to the mesh you have created. When you modify an existing data file, you **must** give the new file a different name from that of the existing file.

2.3 Modifying a data file

If you have chosen to modify the file you have created (or if you have answered `n` to the 'Create a new file?' prompt and selected a `.dat` file from the directory), the existing mesh will be displayed, and by clicking on 'Continue' you will see the **Main Menu** of modification options. These are:

1. *Modify mesh*
2. *Double mesh*

3. *Refine mesh*

4. *Triangulate*

The Text window will also disappear at this point, and all subsequent interaction with the user will be through dialogue windows.

You choose an option by left-clicking on it. If you right-click on a menu option, a Help screen appears describing the option. The facilities in each of these options will now be described.

2.3.1 *Modify mesh*

If you are modifying a datafile written at Basic level, you will be asked if you want to write the new file at Advanced level. If you do want to add Advanced features to the data, you should make this change first. In this case, there are two new facilities offered:

- you will be invited to specify extra mesh parameters (up to four reals and four integers) which will be written with the basic mesh data (NELEM, NPOIN, NDOFN, NPROS) at the start of the datafile. Your 'main engine' may for example require the value of the gravitational constant or the pore water bulk modulus, which can be written here.
- you will be asked if you want to add additional degrees of freedom. There may be a maximum of four degrees of freedom (d.o.f.s) per node. You can also have d.o.f.s defined only at corner nodes of quadratic elements. For example, the CONSL.FOR program models the saturated soil using 8-noded quadrilaterals with (u, v) displacement d.of.s at the four midside nodes and (u, v, p) d.o.f.s at the corners. We would want to increase to 3 the number of d.o.f.s at corner nodes of soil elements only; to avoid applying this to nodes of structural elements, you are prompted to say which element material type LMTYPE the new d.o.f.s should be created for.

You will then be offered the following menu:

1. *Delete existing elements*

2. *Add new elements*

3. *Change nodal coordinates*

4. *Alter constraints* (*i.e.* nodal boundary conditions.) You can also change the number of degrees of freedom at an individual node, here.

5. *Continue*

These options work graphically: you choose an option, then click on the node or element to be processed. In most of the options, you are able to zoom in on the part of the mesh you wish to modify.

To add a new element, you specify its type by `NSIDE` and `NNODE`, then list the nodes around it anticlockwise. You will be prompted for the coordinates and fixity code of any new nodes. If they are midside nodes of the element their coordinates will be calculated by linear interpolation from the corner nodes of the side if you give their coordinates as 0,0, thus making the side straight.

When deleting elements, note that a maximum of 20 elements can be deleted in one batch, but there is a further limitation that not more than 50 nodes can thus become redundant and be deleted also. It is therefore best to delete only a few elements at a time. When nodes are no longer used in the mesh, they are deleted and the remaining nodes automatically renumbered. You can choose an individual element to delete, or you can delete a group of elements by clicking the right mouse-button first, then by clicking on the vertices of a polygon enclosing the group, as described in the next paragraph.

If you have created new degrees of freedom in the mesh, they will not have any boundary conditions associated with them. You can define Dirichlet nodes using 'alter constraints'. For example, if the d.o.f.s have been increased from (u, v) to $u, v, p)$, clicking on the appropriate node will show a three-digit fixity code; the first two digits give the fixities of the original u, v d.o.f.s, and you can change the third digit from 0 to 1 to make the node lie on a Dirichlet boundary for pore-pressure or temperature.

When moving a node (changing its coordinates), you can click on the new position on the graph (zoomed if necessary). The coordinates of the point you have clicked are displayed, and you can edit these to fix the final position of the node. Note that if you wish to move a node beyond the window limits in a zoomed display, you will need to move it to the edge, then restore the full mesh, and zoom again if needed for the further move.

Clicking on 'Continue', you are given the chance to alter the values in the material property sets, including adding new sets (the Text window will reappear for this purpose), and then to change the assignment of elements to property sets, graphically. In this last task, you can click on the individual elements to be reassigned, or **select a group of elements** by enclosing their centres of gravity in a polygon; this is done by first right-clicking anywhere in the mesh, then left-clicking to define each of the vertices around the polygon. Right-click anywhere on the mesh again to join the last vertex to the first, completing the polygon. The elements whose centres lie within the polygon will be reassigned to the specified property set. This facility for defining groups of elements is used again later, in body force loading, for example.

The following Main Menu items allow the user, with a few mouse-clicks and keystrokes, to generate a complex mesh with hundreds of elements and nodes from his/her coarse mesh. Note that when the mesh is refined or doubled, the element material types and fixities get propagated to the newly-created nodes and elements; extra degrees of freedom also get assigned according to the rules that have been specified.

2.3.2 *Double mesh*

Doubling a mesh produces a new mesh with twice the number of elements. The new half of the new mesh is created either by reflecting the existing mesh in a horizontal or vertical line (doubling by reflection) or by joining a copy of the mesh onto one side (doubling by translation). You need to select the type of doubling, and then to select the direction: for example, choosing the positive x-direction will produce a new half of the mesh joined to the existing half on its right-hand side. Node and element numbering in the pre-existing half of the mesh are unchanged by this process.

2.3.3 *Refine mesh*

You can choose to refine the mesh Automatically or Manually. Under Automatic refinement, each quadrilateral element will be divided into 4 quadrilaterals, and each triangular element will be divided into 3 triangles. The new elements and nodes are numbered, but will need to be re-numbered before use in order to reduce the bandwidth of the global stiffness matrix – see below.

Manual refinement only works for quadrilateral elements. Thus, to prepare a complex mesh for Poisson problems, it is most convenient to create the mesh from linear quadrilaterals, refine it using this option, and finally to subdivide each quadrilateral into two linear triangles using the *Triangulate* option. You have the opportunity to zoom to the area of interest before applying refinement; note that the refinement will still be propagated throughout the mesh as appropriate, not just in the zoom region.

Manual division of a quadrilateral element is performed by first specifying an edge by clicking on its corner nodes. Then dialogue windows will appear asking you the number of divisions, whether you want equal divisions, and – if you want unequal divisions – asking you the relative weights (proportional lengths) for each division. Be careful that you are entering the weights in the correct order, from the first end-node to the second. The weightings are relative only; typing 1,2,4,8 or 0.5,1.0,2.0,4.0 will have the same effect.

These divisions will then be used to subdivide the element, and this subdivision will be continue into neighbouring elements, and then throughout the mesh to ensure its consistency. Allocation of fixity codes for the new nodes, and property sets for the new subelements, is performed automatically.

Now a new edge can be chosen and subdivided. Note that there is a restriction on the number of individual element edges in a mesh in this option, so don't try anything too fancy! Also, it is better to subdivide a coarse mesh and then double the refined mesh, rather than doubling first and then refining, to avoid the risk of hitting this maximum on the number of edges.

Although this manual subdivision routine is restricted to quadrilateral elements, it is able to cope with linear and quadratic sides appearing in the same mesh, and will also subdivide one-dimensional and infinite elements (although the 'infinite' sides of infinite elements cannot be subdivided).

2.3.4 *Triangulate mesh*

With this option, each quadrilateral element will be cut into two triangles — a linear quadrilateral will produce linear triangles, and an 8-noded quadrilateral will produce quadratic triangles (with a new node created at the centrepoint). At the Basic level, the whole mesh will be triangulated, with no input from the user. At Advanced level, you can triangulate the whole mesh, a single element, or a group for elements (by defining a region around them using the right mouse-button, as described above).

2.3.5 *Save mesh, and Restart from saved mesh*

The user should periodically save the current mesh to the output file, by clicking on the Save Mesh option. This will overwrite any previously-saved mesh. Should you then make a mistake in, say, refining the mesh, you can return to the main menu and click the Restart from Save option. This reads back the last-saved mesh, and you can re-start from this point.

2.4 Reordering the elements and nodes

When the modifications of the previous section have been completed, the new mesh will look to be ready for use in the finite element analysis. However, it will be very inefficient, since the numbering of the nodes and/or elements will create a global stiffness matrix with a very large bandwidth or frontwidth. (For direct solvers such as Choleski decomposition, the bandwidth depends on the node numbering; for a frontal solver the frontwidth depends on the element numbering.)

You should therefore answer `y` to the prompt to re-order the elements. The reordering algorithm (a simple Cuthill-McKee algorithm) works by asking you to pick (with the mouse) an element at which to start the renumbering; this becomes element 1, then

the neighbouring elements are numbered 2,3,..., and then their neighbours are renumbered, and so on through the mesh. The fronwidth of the resulting renumbered mesh is calculated, and if this is smaller than the frontwidth of the existing mesh, this renumbering is retained. You then have the chance to pick another element at which to start the renumbering, and after each such trial it is the renumbering producing the smallest frontwidth so far which is retained. When you are satisfied that you have achieved a well-numbered mesh, click on 'End'.

This has renumbered the elements, but not the nodes. There is now the option of renumbering the nodes as well; first the nodes around the new element 1 are numbered 1,2,3,..., then the as-yet-unrenumbered nodes around element 2, and so on. The consequent reduction in bandwidth is reported, and there is an option of **displaying graphically the structure of the global stiffness matrix before and after renumbering**. This is an extremely helpful feature for students needing to understand the relationship between node-numbering and stiffness matrix structure.

For elasticity problems, the loading régime has to be added, and you will be prompted whether you wish to proceed to this, or to save the new file without adding loading data. For Laplace and Poisson problems, you will be invited to add data defining the values of u on the Dirichlet part of the mesh boundary.

2.5 Part B: Loading data, for elasticity and frames

This and the next section describe the Part B data which is now required to produce a complete data file for input to a 'main engine.

In Basic-level elasticity and frame analyses, there are two types of loads which can be applied: point loads at nodes, and surface tractions on element edges. In Advanced-level elasticity analyses there are two further types of loads available: specified displacements at nodes, and body forces over elements. As in the 'Modify mesh' option, the user is able to zoom on the part of the mesh to be loaded, before applying each type of load.

To apply a **point load** at a node, click on the node and then type in the components of the force in the x- and y-directions. (A negative value of the component means that it acts in the negative x or y direction) The same process is used to apply **specified displacements** at nodes, at Advanced level.

At Advanced level, **body forces** can now be input. The components of the force are typed in, and then the user clicks on the elements to which this force applies, or defines a group of elements by clicking on the vertices of a bounding polygon as described earlier.

Surface tractions along element edges are entered by first creating surface traction sets defining the normal and tangential the components of the pressure acting at end nodes of the edge; each such pressure is allocated a traction set number. The first

component of the pressure is the component acting normal to the edge; a positive value means that it acts onto the element, compressing it. The second component is the tangential component of the pressure, *i.e.* acting along the edge; a positive sign means that it acts anticlockwise around the element. Note that these are spot values of the pressure field (typically in Kilonewtons per square metre) distributed over the edge. In the case of line elements (representing trusses or beams), the 'body' of the element is imagined as if the node numbering continued, anti-clockwise, around a two-dimensional element with the line as the first edge.

Once the surface traction sets have been entered, they are assigned to nodes defining the edges on which those sets act. First click on an element with loaded side. Then click on the two corner nodes defining the edge to be loaded. Then type the surface traction sets corresponding to each corner node. If the same set is assigned to each corner node, it means that a uniform surface traction with this value is applied along the edge. If two different sets are assigned, the surface tractions change linearly from one value to the other along the side. If you have only defined one traction set, this will automatically be assigned as a uniform load on each loaded edge.

2.6 Part B: Boundary loading data

The next block of data, defining various types of boundary 'loading', is used for Poisson analysis and advanced elasticity problems. For Basic-level elasticity and beam analyses, a header for the variables `nbdry` (no. of boundary data sets) and `nbdir` (no. of Dircihlet boundary planes), followed by a blank line (i.e. `nbdry = nbdir = 0`) is written in the data file.

The nature of the first set of boundary loading depends on the application: for Poisson problems (NDOFN=1) it defines radiating boundaries, while for advanced elasticity problems (NDOFN=-2) it defines 'unloading' boundaries associated with excavation-type loading; the coding for this type of loading is included in ELADV.FOR.

The second set of boundary 'loading' data defines planes on which the unknown potential field takes a specified value, i.e. Dirichlet boundaries. This is important in Poisson-type problems (NDOFN=1), and is also used for the scalar temperature field in the coupled elasticity program THERM.FOR (NDOFN=-2).

We first deal with **radiating boundaries**, for Poisson problems (NDOFN=1). Here, the boundary condition is

$$q_n = \bar{q} + \alpha U \tag{2.2}$$

where q_n is the normal flow (gradient of U normal to the boundary), $\bar{q}$ is a specified flow, and α is a radiation constant. Up to nine sets of radiation boundaries can be

entered. The values of $\bar{q}$ and α are entered, and then you click on the nodes in the mesh associated with this boundary condition set.

For Advanced elasticity problems you will be asked if you wish to define **excavation boundaries**; these are only used in conjunction with program ELADV.FOR (or your own 'main engine' if you have included this feature in it). First, an in situ stress field can be prescribed. The type of stress field is decided by ISTYP, read from the first dataline in this section. The two parameters defining the stress field are also read from this line, and are:

`ISTYP`	=	Stress field type (1 or 2)
`ISET`	=	Set no.
`REAL1`	=	horizontal stress σ_x (ISTYP=1) or soil density γ (ISTYP=2)
`REAL2`	=	vertical stress σ_y (ISTYP=1) or lateral stress ratio K_0 (ISTYP=2)

Thus, for ISTYP=1 the stress at all Gauss points of soil elements (with material type $LMTYP(L) > 1$) is (σ_x, σ_y). For ISTYP=2, the stress state at a soil element Gauss point with coordinates $(x, y), y < 0$ is $(-K_0\gamma y, -\gamma y)$. Note that in ELADV, material type LMTYPE=1 is assumed to mean the element is a structural element with no initial stresses; soil/rock elements have LMTYPE=2,3,.... Also, with an overburden stress field, the plane $y = 0$ is taken as the ground level. See the description of the program ELADV.FOR, for further information.

Finally, for Poisson problems or thermoelasticity analyses, you will be invited to input data defining the values to be assigned to the variable U on the Dirichlet part of the mesh boundary, which was specified by the fixity codes. (Parts of the boundary which have not been given such fixed values, will have a natural Neumann boundary condition – e.g. insulated, in a heatflow application). This data section is also offered when preparing vector-variable meshes at Advanced level, since you may have created extra degrees of freedom for which scalar-variable-type boundary conditions are needed (the temperature field in thermoelasticity, for example). At Advanced level, you are therefore asked which degree of freedom each boundary condition set applies to. Note that this data is only handled in POISS.FOR and THERM.FOR.

You specify **Dirichlet boundary values** by defining a series of planes, horizontal or vertical, and for each plane assigning a value of u which all Dirichlet nodes which lie on that plane, will take. A vertical plane is defined by specifying the x-coordinate; a horizontal plane is defined by specifying the y-coordinate. In the text screen, you are prompted to type x or y, then the coordinate value, and finally the value of u associated with that plane.

The program `POISS.FOR` is written to accept values for up to 8 such planes. The conditions will be applied in order, so that if a point lies on the intersection of two planes, it is the later condition which will apply.

For any Dirichlet nodes whose value has not been fixed by this method, the program `POISS.FOR` will use its in-built boundary value function `bdryfn(x,y)` to assign values

at the Dirichlet nodes. The fixed-planes and the boundary-function techniques can thus be used in combination, to produce a complex boundary régime.

2.7 Editing existing Part B data

If the input datafile already contains Part B data, the user may opt to edit this rather than create completely new Part B data. If the editing option is chosen, a warning is given that "this will only make sense if you have not changed the mesh topology!. That is, in processing the Part A data earlier, any deletion of elements/nodes, or mesh doubling or refinement, will have resulted in renumbering of the elements/nodes, so that the load data in the input file will refer to elements/nodes whose numbers have changed.

Each applied load will be presented separately to the user, who can choose whether to include it in the new data file or to ignore it. There is also scope to edit it, by changing the load magnitude, for example, before including; additional new loads can also be specified. The locations of the elements or nodes referred to in each data item will be displayed on the mesh graph, so the user can see if the item still applies to the correct location. In the display of elements associated with a particular body force set, displayed elements can be deleted from the set by clicking on them, and new ones added. Part B data items which no longer make sense — element/node numbers which now exceed NELEM/NPOIN, or surface tractions on elements where the node numbers no longer define an element edge — will automatically be discarded. This load modification process also works with any boundary set or Dirichlet plane data from the original datafile.

2.8 Exiting the pre-processor

At Advanced level, you are also invited to provide data specifying incremental loading and timestepping regimes - see below.

Once all the Part B data have been added, the pre-processor reports that the new data file has been written. The Part B data in a readable format will be appended to the print file (with a `.prt` filename extension) which contains the input data in readable form. You can also generate a PostScript graphics file of the mesh, which will have a `.ps` filename extension. Then you can exit the pre-processor, or return to modifying this or another data file. If you select a data-file to modify (usually the one which has just been written), the Text window will reappear.

2.9 Format of the final data file

In the `.dat` data file produced by the pre-processor, there are one-line headers before each block of data — these are ignored by the main program. The data file has the following block structure:

Block	Data type
1	Title
2	Basic mesh constants
3	Element data: element-node connections, and property set
4	Nodal data: fixity code and coordinates
5	Material properties

The format of each block is as follows:

Block	Format	Variables
1	A50	`TITLE` (max. 80 characters)
2	4I5	`NPOIN NELEM NDOFN NPROS`
3	I1,I4,I2,I3,9I5	`LCARD NSIDE LMTYPE LSET L (LNODS(L,N),N=1,NNODE)`
4	I1,I4,I5,2G10.3,I5	`LCARD KODE NP COORD(NP,1) COORD(NP,2) NDOF(NP)`
5	I1,I4,I5,10X,G10.3	`LCARD NSET ICOMP VALUE`

where

`NPOIN` = No. of nodes in mesh
`NELEM` = No. of elements in mesh
`NDOFN` = No. of degrees of freedom per node, see §2.3
`NPROS` = No. of material property sets
`NSIDE` = No. of sides in element L
`LMTYPE` = Material type of element L
`NNODE` = No. of nodes in element L
`LSET` = Material property set number of element L
`LNODS(L,N)` = Global node number of the N'th node of element L
`KODE` = Fixity code of node NP; see above
`COORD(NP,I)` = I'th coordinate of node NP
`NDOF(NP)` = No. of degrees of freedom of node NP
`VALUE` = value of component ICOMP of material property set NSET

`LCARD` is a character which indicates the last line of a data block by holding the value 1; otherwise it is zero or blank.

At Advanced level, extra mesh constants (two reals, four integers) will be appended to block 2, so this line will have format 4I5,2G10.3,4I5.

Note that all variable names follow the Fortran naming convention: integers begin

with a character in the range I–N.

For vector-variable problems, the mesh data is followed by data lines describing the applied loading. This data contains the following blocks (separated by header lines):

Block	Data type
L1	Number of body force and surface traction sets
L2	Nodal forces and specified displacements
L3	Body force sets
L4	Elements in each body force set
L5	Surface traction sets
L6	Edges with applied surface tractions

The format of each block is as follows:

Block	Format	Variables
L1	2I5	`NBSET,NPSET`
L2	I1,I4,I5,2E10.3	`LCARD LD NP FX FY`
L3	I1,I4,3X,I2,10X,E10.3	`LCARD ISET I12 BXY`
L4	I1,I4,14I5	`LCARD,ISET,`elements with body force set ISET
L5	I1,I4,3X,2I2,8X,2E10.3	`LCARD NSET I1 I2 PN PT`
L6	I1,I4,I5,2(I5,I2,1X)	`LCARD NNAS L N1 NSET1 N2 NSET2`

where:

`NBSET` = No. of body force sets
`NPSET` = No. of surface traction sets
`LD` = 1 for nodal force; 2 for specified displacement
`FX,FY` = Forces in x and y directions at node NP
`BXY` = Body force in x (if I12=1) or y (if I12=2) direction in body force set ISET
`I1,I2` not used
`PN,PT` = Normal & tangential components of surface tractions in set NSET
`N1,N2` = corner nodes of element L defining a loaded edge
`NNAS` = Number of nodes (2 or 3) along edge of element L
`NSET1,NSET2` = surface traction sets associated with nodes N1 and N2

Blocks L3 and L4 do not appear in Basic-level data sets, and otherwise only appear if NBSET>0. Similarly, blocks L5 and L6 only appear if NPSET>0.

In block L5, the variables I1 and I2 take the values 1 and 2 respectively, to indicate that the following two values are the normal and tangential components of the traction, respectively.

Warning: in versions of FELIPE before v3.1, block L6 was written in format I1,I4,I5,2(I3,I2).
This has been changed to cater for node numbers greater than 999 in large refined meshes.

If you try to modify a data-set produced by an earlier version of FELIPE, loading data which no longer makes sense when read in the new format will be ignored.

For Poisson problems (and for vector-variable files written at Advanced level, i.e. for files with NDOFN less than 2), this data is followed by data lines describing the boundary conditions for scalar d.o.f.s. A header is followed by a line (format 2I5) stating the number of radiating (for NDOFN=1) or excavation (NDOFN=-2) boundary sets – parameter `NBDRY` and the number of Dirichlet boundary planes – parameter `NBDIR`. As with NBSET and NPSET above, if either of these parameters is zero then the corresponding block described below does not appear.

After a header line, radiating boundary sets (if NDOFN=1 and NBDRY is not 0) are defined in lines of the form:

LCARD,INDOF,ISET,ALPHA,QBAR, nodes in this set

in format I1,2I2,2E10.3,10I5, where

`INDOF` = Degree of freedom to which this condition applies (default 1)
`ISET` = Set no.
`ALPHA` = radiation constant
`QBAR` = Specified normal flow

In the case of excavation boundaries (NDOFN=-2 and NBDRY not 0), the post-header lines are of the form:

LCARD,ISTYP,ISET,SX,SY, nodes in this set (if ISTYP=1: uniform stress field)

or

LCARD, ISTYP,ISET,GAMMA,K0, nodes in this set (if ISTYP=2: overburden stress field)

in format I1,2I2,2E10.3,10I5. See §2.6 for the definition of these variables.

The second block in this section, if NBDIR is not 0, defines Dirichlet boundary values (see §2.6 above). The data for the value of u on each plane defined by an x (for vertical plane) or y (for horizontal plane) coordinate, is given in a line:

LCARD,component,‘ =’,COORD,VALUE,INDOF

where component is either ‘x’ or ‘y’, in format: I1,A2,A2,E10.3,5X,E10.3,I5. INDOF is the degree of freedom to which the condition applies. For example, the line:

0 x = 4.000 60.000 1 means that $U = 60.0$ for Dirichlet nodes on the plane $x = 4.0$,

where U is the 1st degree of freedom at the node.

Note that in all FELIPE programs, the standard Fortran naming convention (integers start with characters between I and N) is followed.

2.10 Incremental loading and timestepping régimes

There are two further lines of loading data which may be added if the file is being prepared at Advanced level. These define an incremental loading régime and a time-stepping régime respectively:

Nonlinear analyses normally involve adding the loading in increments. Thus once the total loads have been specified, the user is asked at Advanced level if these should be added in increments. The user specifies the number of increments (maximum 9), and then the size of each increment; this data is appended to the data file in a line of format I5,9F5.2 .

Parabolic p.d.e.s, and time-dependent material models such as elasto-viscoplasticity and poroelasticity, require a timestepping algorithm in their 'main engines', and the data for this can be input as the final section of load data at Advanced level. The parameters which can be specified are:

1. Initial timestep size DTINT - default 0.001
2. Timestep increase factor FTIME - default 1.2
3. Max. allowable timestep increase ALLOW - default 10.0
4. Time discretization parameter THETA - default 0.0
5. Final time TTEND - default 100.0
6. Timestep for initial solution DTONE - default 0.0

The purpose of these parameters is to define the following timestep control algorithms:

2.10.1 Algorithm used in VPLAS.FOR:

The timestepping will start with a timestep of Δt_0 =DTINT, the next timestep will be Δt_1 =DTINT*FTIME, and thereafter the timestep will be increased by a factor of FTIME at each iteration (unless this is overridden by a timestep control algorithm within

the 'main engine' itself). When the timestep would be greater than DTINT*ALLOW, it is capped at Δt =ALLOW. Whenever a new load increment is added, the timestep reverts to Δt =DTINT.

The parameter THETA, $0 \leq \theta \leq 1$, defining the time discretization algorithm is a common feature of many schemes; $\theta = 0.0$ defines an explicit discretization, $\theta = 1.0$ defines a fully-implicit scheme.

TTEND and DTONE are not used in the viscoplasticity timestepping régime.

2.10.2 Algorithm used in CONSL.FOR

For soil consolidation, an initial solution is obtained using the timestep DTONE - for the two-level element in the CONSL.FOR program, a value of zero is acceptable for this. Then, the timestep DTINT is used for five timesteps before it is increased by a factor of FTIME, for a further five steps, and so on. Timestepping halts when the time TTEND is reached, or earlier if requested by the user at run-time. THETA is also used in this program; a fully-implicit scheme where THETA=1, is recommended.

The timestepping data DTINT,FTIME,ALLOW,THETA,TTEND,DTONE are written in a line with format 6E10.3.

Of course, you are free to modify the timestepping régimes in the 'main engines' to your particular needs, and recompile the programs.

Chapter 3

The Poisson program `POISS.FOR`

3.1 Theory

The following is only a brief summary of the relevant equations; you should refer to a finite element textbook or lecture notes for a proper description.

3.1.1 Problem definition

The Poisson equation to be solved over a region in the (x, y) plane is

$$-a_x \frac{\partial^2 u}{\partial x^2} - a_y \frac{\partial^2 u}{\partial y^2} = f(x, y) \tag{3.1}$$

where the parameters a_x, a_y will be constant over each element. On the Dirichlet part of the boundary, the value of u will be specified by

$$u(x, y) = g(x, y) \quad \text{on} \partial\Omega_D,$$

and on the Neumann part of the boundary, the condition is of no outward normal flow, that is

$$\frac{\partial u}{\partial n} = 0 \quad \text{on} \partial\Omega_N.$$

The right-hand-side function $f(x, y)$, as well as the function $g(x, y)$ defining the values of u on the Dirichlet boundary, will be included in the program `POISS.FOR` as function routines.

The Laplace equation is a special case of the above, where $a_x = a_y = 1$ and $f(x, y) = 0$, and so will not be considered separately here.

3.1.2 Finite element formulation

The associated bilinear form for (3.1) is:

$$a(u,v) = \iint_\Omega \{a_x \frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + a_y \frac{\partial u}{\partial y}\frac{\partial v}{\partial y}\}\, dx\, dy, \tag{3.2}$$

and the variational problem which will produce solutions of (3.1), is to minimize the functional

$$I(u) = a(u,u) - 2\iint_\Omega f.u\, dx\, dy. \tag{3.3}$$

For full details of the theory, see a standard text on the finite element method.

Consider now the body Ω discretized by a mesh of 3-noded linear triangle elements, and consider a typical element e having vertices at $(x_1,y_1),(x_2,y_2)$ and (x_3,y_3) in the (x,y) plane. We will map this to the canonical element with vertices $(0,0),(1,0)$ and $(0,1)$ in the (ξ,η) plane, this being the local coordinate system. The mapping is achieved by

$$x = x_1 + (x_2 - x_1)\xi + (x_3 - x_1)\eta \tag{3.4}$$

and

$$y = y_1 + (y_2 - y_1)\xi + (y_3 - y_1)\eta. \tag{3.5}$$

Now we approximate the value of the unknown variable $u(x,y)$ at a point within the element, as a linear combination of the nodal values U_i and the shape or basis functions $\phi_i(x,y)$:

$$u(x,y) = U_1\phi_1(x,y) + U_2\phi_2(x,y) + U_3\phi_3(x,y). \tag{3.6}$$

The basis functions, which must have the property that $\phi_i(x_j,y_j) = \delta_{ij}$, will be defined in (3.16-18) below.

Then the finite element system to be solved, obtained by setting

$$\frac{\partial I}{\partial U_i} = 0,\ i = 1,2,\ldots,M$$

from (3.3) is:

$$\sum_{j=1}^{M} a(\phi_j,\phi_i)\, U_j = \iint_\Omega f.\phi_i\, dx\, dy \quad i = 1,2,\ldots,M \tag{3.7}$$

with boundary conditions

$$U_i = g(x_i,y_i) \quad \text{for}(x_i,y_i) \in \partial\Omega_D.$$

Collecting the unknown nodal variables in the vector U, we write (3.7) as the linear system

$$\mathbf{K}U = F \tag{3.8}$$

to solve for U. From (3.2), the (i, j)'th entry in the 3×3 element stiffness matrix $\mathbf{K}^e$ for element e, is

$$K_{ij}^e = \iint_{\Omega_e} \{a_x \frac{\partial \phi_i}{\partial x} \frac{\partial \phi_j}{\partial x} + a_y \frac{\partial \phi_i}{\partial y} \frac{\partial \phi_j}{\partial y}\} \, dx \, dy, \tag{3.9}$$

and the contribution from element e to the right-hand-side force vector has i'th component

$$F_i^e = \iint_{\Omega_e} f.\phi_i \, dx \, dy. \tag{3.10}$$

3.1.3 Evaluating the Cartesian derivatives

We can relate the derivatives of a typical basis function ϕ with respect to the (x, y) variables, to the local derivatives with respect to (ξ, η), by

$$\begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \frac{\partial \phi}{\partial \eta} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{bmatrix} \tag{3.11}$$

where $\mathbf{J}$ is the 2×2 Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}. \tag{3.12}$$

¿ From the mapping (3.4,3.5), we see that

$$\mathbf{J} = \begin{bmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{bmatrix}. \tag{3.13}$$

Equation (3.11) can be rearranged to

$$\begin{bmatrix} \frac{\partial \phi}{\partial x} \\ \frac{\partial \phi}{\partial y} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \frac{\partial \phi}{\partial \eta} \end{bmatrix}, \tag{3.14}$$

and inverting $\mathbf{J}$ gives

$$\begin{bmatrix} \frac{\partial\phi}{\partial x} \\ \frac{\partial\phi}{\partial y} \end{bmatrix} = \frac{1}{|\mathbf{J}|}\begin{bmatrix} y_3 - y_1 & -(y_2 - y_1) \\ -(x_3 - x_1) & x_2 - x_1 \end{bmatrix}\begin{bmatrix} \frac{\partial\phi}{\partial\xi} \\ \frac{\partial\phi}{\partial\eta} \end{bmatrix}, \tag{3.15}$$

where $|\mathbf{J}| = \det \mathbf{J}$, the determinant of $\mathbf{J}$. This is equal to 2Δ, where Δ is the area of the triangle.

We now introduce the linear basis functions:

$$\begin{aligned} \phi_1(\xi,\eta) &= 1 - \xi - \eta, & (3.16)\\ \phi_2(\xi,\eta) &= \xi, & (3.17)\\ \phi_3(\xi,\eta) &= \eta, & (3.18) \end{aligned}$$

so that, using (3.15) we get the Cartesian derivatives as

$$\begin{aligned} \frac{\partial\phi_1}{\partial x} &= -\frac{1}{|\mathbf{J}|}(y_3 - y_2) & (3.19)\\ \frac{\partial\phi_2}{\partial x} &= \frac{1}{|\mathbf{J}|}(y_3 - y_1) & (3.20)\\ \frac{\partial\phi_3}{\partial x} &= -\frac{1}{|\mathbf{J}|}(y_2 - y_1) & (3.21)\\ \frac{\partial\phi_1}{\partial y} &= \frac{1}{|\mathbf{J}|}(x_3 - x_2) & (3.22)\\ \frac{\partial\phi_2}{\partial y} &= -\frac{1}{|\mathbf{J}|}(x_3 - x_1) & (3.23)\\ \frac{\partial\phi_3}{\partial y} &= \frac{1}{|\mathbf{J}|}(x_2 - x_1). & (3.24) \end{aligned}$$

Integration over the element e, for the right-hand-side vector in (3.10), is achieved by

$$\int\int_{\Omega_e} f.\phi_i \; dx \; dy = \int_0^1 \int_o^{1-\eta} f.\phi_i \; |\mathbf{J}| \; d\xi \; d\eta. \tag{3.25}$$

3.2 Program operation

Note that the program has only basic interaction with the user via the MS-DOS box; this is because the philosophy of FELIPE is to write the 'main engines' as standard Fortran77, which can be compiled on any Fortran77 compiler. No use is made of the language extensions specific to the FTN77 compiler. Of course, you are welcome to modify these

programs to suit your own needs, exploiting the language extensions featured in the compiler you are using.

A number of secondary subroutines used by the Poisson main program `POISS.FOR` are contained in the source code file `FELSB.FOR`. Thus, these two files should be compiled, and then linked to produce the executable file `POISS.EXE`.

When `POISS.EXE` is run, the program will prompt you to enter the name of the `.dat` data file – which should be entered without the .dat extension – and will open this on unit 15 and read it using the subroutine `input`. It will write the input data in readable form, to a file with a `.prt` filename extension, opened on unit 16, using the subroutine `print`. Also, the data is echoed to an output file with `.out` filename extension, opened on unit 17.

During the finite element calculation, a scratch file is opened on unit 18 to store the element stiffness matrices. After performing the finite element calculation, the results are appended to the print file, and the output file, by the subroutine `result`. The `.prt` file can be printed, and the `.out` file is used by the post-processor FELVUE.

3.3 Program structure

The main program performs the finite element calculation by calling the following subroutines in turn:

`stiff` This subroutine calculates the element stiffness matrices and writes them to the scratch file. It also calculates the right-hand side load vector F.

`bndry` This subroutine reads the Dirichlet boundary values from the `.dat` file.

`assmb` This subroutine assembles the element stiffness matrices to form the global stiffness matrix.

`solve` This subroutine solves the global stiffness equation (3.8), using Choleski decomposition.

Some more detail on each of these subroutines is now given. Note that the source code is provided with comment lines explaining the sections, and describing the variables used.

3.3.1 `stiff`

The global right-hand side vector F in (3.8) is prepared in the array `aslod`. For each element, the contribution to this is calculated by (3.10), and the element stiffness matrix

K^e by (3.9). The integrations involved are performed numerically, by the one-point Gauss rule:

$$\int_0^1 \int_o^{1-\eta} F(\xi,\eta)\,|\mathbf{J}|\,d\xi\,d\eta \doteq \frac{1}{2}F(\frac{1}{3},\frac{1}{3}). \tag{3.26}$$

These local coordinates of the Gauss point are stored in `(s,t)`, and mapped to Cartesian coordinates `(xg,yg)` using the mapping (3.4-5). The determinant of the Jacobian matrix J in (3.13) is found, and the basis functions and Cartesian derivatives in (3.16-25) calculated at the Gauss point. The entries in the element stiffness matrix `estif` are formed according to (3.9), and written to the scratch file.

The contribution to the load vector by (3.10) is added in also. For this, the right hand side vector $f(x,y)$ in (3.1) must be specified. This is done in the function `rhsfn(x,y)` to be found in the file `POISS.FOR` after the `result` subroutine. It is programmed as

$$f(x,y) \equiv 1,$$

but you can change this for your own problem.

3.3.2 `assmb`

The element stiffness matrices are read from the scratch file, and assembled to form the global stiffness matrix K in the array `gstif`. As K is a banded matrix, it is stored compactly, only the band itself being stored. First the halfbandwidth `nhbw` is calculated; this is the maximum, over all elements, of the maximum difference between node numbers within the element. Then the lower diagonal of the band only (since K is symmetric) is stored in `gstif`, in a compact block with `npoin` rows and `nband` columns, where `nband = nhbw+1`. The i,j'th entry in K is stored at the location
$K_{ij} \rightarrow$ `gstif(i,j-i+nhbw+1)`.
See Chapter 8 for further details of the compact storage techniques used in FELIPE.

3.3.3 `solve`

This subroutine uses Gaussian elimination to solve (3.8). It calls three subroutines in turn:

`fixdof` This subroutine modifies the stiffness matrix K and right-hand side vector F for rows corresponding to nodes on the Dirichlet part of the boundary. The value of u at such a node is found by comparing the nodal coordinates with those of the Dirichlet planes defined by the data read in `bndry`. The program is dimensioned to store a maximum of 8 such planes. They are defined by

– the component ('x' or 'y') stored as a character in the array `dbdcmp`;

– the coordinate of x or y defining the plane, stored in the array `dbdcrd`;

– the value of u for nodes on that plane, stored in the array `dbdval`.

The variable `ndbdry` stores the number of planes.

For Dirichlet nodes which do not have their value specified by a plane as described above, the subroutine will call the function `bdryfn(x,y)`; this function is programmed as

$$g(x, y) = x^2 - y^2;$$

you should edit this to the function specified in your problem, if you want to define a complex boundary value distribution by this means.

If $U = 0$ at the node i, the entry on the diagonal in the stiffness matrix K is given a large value ($K_{ii} \leftarrow K_{ii} * 10^{20}$), and the i'th entry in F is set to zero; this ensures that in the Gaussian elimination the value of u_i that is calculated will be very small. If $U \neq 0$ at the node i, the diagonal entry in K is multiplied by 10^{12}, and the resulting number, multiplied by the value of U, is substituted for the entry in F.

`chodec` This subroutine performs the Choleski decomposition of K into the factorization $L.L^T$, where L is lower triangular. The entries in L are overwritten over those of the lower band of K in `gstif`.

`chosub` This subroutine solves $K.u = F$ in two stages: solve $L.y = F$ by forward substitution, then solve $L^T.u = y$ by back-substitution. The vector of nodal unknowns U is produced in the array `asdis`. Small values resulting from the Dirichlet 'trick' in `fixdof` are set to zero.

Further details of the matrix storage and solution techniques will be found in Chapter 8.

3.3.4 `result`

This section writes the results — the nodal values of U — into the print file (*filename*`.prt`). It also appends a one-line header to the output file (*filename*`.out`) and then for each node n writes `N` and `UN`, the value of U at node n, in the format `I5, E10.3`. This file will be read by the post-processor.

3.4 Examples

Sample datafiles podemo.dat, pohex2.dat and pohex3.dat are provided in the FELIPE package; these are documented in Chapter 9.

Chapter 4

The elasticity program ELAST.FOR

4.1 Theory

The program is written to perform a finite element analysis of an elastic body, which can be formed by a combination of different materials, constrained to deform in plane strain under a prescribed load. The body Ω, with loaded boundary Γ, is discretized by a mesh of eight-noded isoparametric quadrilateral elements.

Only a brief summary of the theory is provided here; you should consult a finite element text, such as *The Finite Element Method* by O. C. Zienkiewicz, for a full description – see Section 9 for recommendations.

4.1.1 Stresses and Strains

Let the stress and strain at a point in the body in plane strain, omitting the out-of-plane direction, be given by the vectors

$$\vec{\sigma} = (\sigma_x \quad \sigma_y \quad \tau_{xy})^T$$

and

$$\vec{\epsilon} = (\epsilon_x \quad \epsilon_y \quad \gamma_{xy})^T$$

respectively.

Let $\vec{\mathbf{u}} = (u \quad v)^T$ be the displacement at the point $(x \quad y)^T$.

In a continuous body, strain at a point is the rate of change of displacement with distance, ie. the direct strains in the x and y directions are given by

$$\epsilon_x = \frac{\partial u}{\partial x} \qquad \text{and} \qquad \epsilon_y = \frac{\partial v}{\partial y}. \tag{4.1}$$

The shear strain is given by

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}. \tag{4.2}$$

These relationships may be expressed by the matrix equation

$$\vec{\epsilon} = \mathbf{A}\vec{\mathbf{u}} \tag{4.3}$$

where $\mathbf{A}$ is the strain-displacement operator

$$\mathbf{A} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}. \tag{4.4}$$

According to the linear elasticity constitutive law, stresses and strains are related linearly by

$$\vec{\sigma} = \mathbf{D}\vec{\epsilon} \tag{4.5}$$

where $\mathbf{D}$ is the constitutive matrix. In plane strain (that is, there is no movement in the out-of-plane direction), this matrix takes the form

$$\mathbf{D} = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{4.6}$$

where E is Young's modulus and ν is Poisson's ratio.

In plane stress (that is, the out-of-plane stress is zero), this matrix takes the form

$$\mathbf{D} = \frac{E}{(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1}{2}(1-\nu) \end{bmatrix}. \tag{4.7}$$

You choose the plane strain or plane stress by setting the variable mstyp to 1 or 2 respectively when prompted at the start of the main program.

For an isoparametric quadrilateral element, there are eight nodes. The nodal displacement vector $\vec{\mathbf{u}^e}$ for a single element will therefore be

$$\vec{\mathbf{u}^e} = (u_1 \quad v_1 \quad u_2 \quad v_2 \quad u_3 \quad v_3 \quad \dots \quad u_8 \quad v_8)^T. \tag{4.8}$$

The basis functions (or shape functions) $\{N_i,\ i = 1, 2, \dots, 8\}$ are used to interpolate for the displacement at a point within the element:

$$u(x, y) = \sum_{i=1}^{8} N_i(\xi, \eta)u_i, \quad v(x, y) = \sum_{i=1}^{8} N_i(\xi, \eta)v_i$$

and then by (4.3) the strain vector $\vec{\epsilon}$ at a point (x, y) will be given by

$$\vec{\epsilon} = \mathbf{B}\vec{\mathbf{u}^e} \tag{4.9}$$

where

$$\begin{aligned} \mathbf{B} &= \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \cdots & \frac{\partial N_8}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & \cdots & 0 & \frac{\partial N_8}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \cdots & \frac{\partial N_8}{\partial y} & \frac{\partial N_8}{\partial x} \end{bmatrix} \\ &= \mathbf{AN} \end{aligned} \tag{4.10}$$

where

$$\mathbf{N} = \begin{bmatrix} N_1 & \cdots & N_8 \\ N_1 & \cdots & N_8 \\ N_1 & \cdots & N_8 \end{bmatrix}.$$

Combining equations (4.5) and (4.8), we have

$$\vec{\sigma} = \mathbf{DB}\vec{\mathbf{u}^e} \tag{4.11}$$

4.1.2 Principle of Minimum Total Potential Energy

The total potential energy of a system is defined as

$$\Phi = U + W \tag{4.12}$$

where W is the potential energy of the external forces in the deformed configuration, and is defined as

$$W = -\iint_\Omega \vec{\mathbf{u}}^T \vec{\mathbf{b}} dV - \int_\Gamma \vec{\mathbf{u}}^T \vec{\mathbf{p}} dS \tag{4.13}$$

and U is the strain energy of the deformed structure, and is given by

$$U = \frac{1}{2} \iint_\Omega \vec{\epsilon}^T \vec{\sigma} dV. \tag{4.14}$$

Substituting equations (4.8) and (4.10) into equation (4.13), we obtain

$$U = \frac{1}{2} \sum_e \iint_\Omega \vec{\mathbf{u}^e}^T \mathbf{B}^T \mathbf{D} \mathbf{B} \vec{\mathbf{u}^e} dV \tag{4.15}$$

The potential energy of the external forces may be simplied if we can approximate the surface and body forces by a set of equivalent nodal forces $\vec{\mathbf{q}}$. Thus, the force at any point in an element would be given by $\sum_{i=1}^{8} N_i q_i$, which we write as $\mathbf{N}^T \vec{\mathbf{q}}$, when

$$\mathbf{N} = \begin{bmatrix} N_1 & N_2 & \cdots & N_8 \\ N_1 & N_2 & \cdots & N_8 \end{bmatrix}.$$

Then $W = -\iint_V \vec{\mathbf{u}^e}^T \mathbf{N}^T \vec{\mathbf{q}} dV$,

so that the total potential energy of the system is

$$\Phi = \frac{1}{2} \iint_V \vec{\mathbf{u}^e}^T \mathbf{B}^T \mathbf{D} \mathbf{B} \vec{\mathbf{u}^e} dV - \iint_V \vec{\mathbf{u}^e}^T \mathbf{N} \vec{\mathbf{q}} dV \tag{4.16}$$

By variational principles, this is minimized when

$$\frac{\partial \Phi}{\partial \vec{\mathbf{u}}} = 0,$$

where $\vec{\mathbf{u}}$ is the global vector of nodal displacements. Applying this to (4.15), we obtain the system of equations

$$\frac{\partial \Phi}{\partial \vec{\mathbf{u}}} = \iint_V \mathbf{B}^T \mathbf{D} \mathbf{B} \vec{\mathbf{u}} dV - \iint_V \mathbf{N}^T \vec{\mathbf{q}} dV = 0$$

or

$$\mathbf{K} \vec{\mathbf{u}} = \vec{\mathbf{f}} \tag{4.17}$$

where

$$\mathbf{K} = \iint_V \mathbf{B}^T \mathbf{D} \mathbf{B} dV \qquad \text{(stiffness matrix)} \tag{4.18}$$

and

$$\vec{\mathbf{f}} = \iint_V \mathbf{N}^T \vec{\mathbf{q}} dV \qquad \text{(consistent load vector)} \tag{4.19}$$

4.1.3 Element stiffness matrices

As in the Poisson theory of the previous chapter, the local and Cartesian derivatives of the basis functions are related by

$$\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} = \mathbf{J} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} \tag{4.20}$$

where $\mathbf{J}$ is the 2×2 Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}. \tag{4.21}$$

As the element we shall be using is isoparametric, we can express the coordinates at a point, in terms of the nodal coordinates by

$$x = \sum_{i=1}^{8} N_i x_i, \quad y = \sum_{i=1}^{8} N_i y_i,$$

in the same way as for the displacements, where (x_i, y_i) are the global coordinates of node i. Thus,

$$\frac{\partial x}{\partial \xi} = \sum_{i=1}^{8} \frac{\partial N_i}{\partial \xi} x_i, \qquad \text{etc.} \tag{4.22}$$

Equation (3.3) can be rearranged to

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} \tag{4.23}$$

so that the 2×2 matrix $\mathbf{J}$ will need to be inverted, if we are to find the Cartesian derivatives.

From equation (4.17), we have

$$\mathbf{K}^e = \iint_{\Omega^e} \mathbf{B}^T \mathbf{D} \mathbf{B} \; T \; dx \, dy.$$

where T is the material thickness, given as the third material property component for ELAST.FOR and ELADV.FOR. In what follows, we assume $T = 1.0$, for simplicity.

As we want to work with the $\xi\eta$-coordinate system, we have

$$\iint_{\Omega^e} F\,dx\,dy = \int_{-1}^{1}\int_{-1}^{1} F \det \mathbf{J} \cdot d\xi\,d\eta \tag{4.24}$$

where $\det \mathbf{J}$ is the determinant of the Jacobian matrix.

The element stiffness matrix then becomes

$$\mathbf{K}^e = \int_{-1}^{1}\int_{-1}^{1} \mathbf{B}^T\mathbf{D}\mathbf{B}(\det \mathbf{J})d\xi\,d\eta. \tag{4.25}$$

To perform the numerical integration, the following 2×2 Gaussian integration rule is used:

$$\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)d\xi\,d\eta \doteq \sum_{i=1}^{4} w_i f_i \tag{4.26}$$

where $w_1 = w_2 = w_3 = w_4 = 1$, and f_i are the values of f at the four Gauss-points $(\pm\frac{1}{\sqrt{3}}, \pm\frac{1}{\sqrt{3}})$.

When the stiffness matrix for each element is found, we can then assemble them to form a global stiffness matrix.

4.1.4 The Load Vector

We have defined the equivalent nodal load vector as

$$\vec{\mathbf{f}} = \iint_V \mathbf{N}^T\vec{\mathbf{q}}\,dV$$

where $\mathbf{N}$ is a matrix containing the basis functions, and

$$\vec{\mathbf{f}} = (F_{x1} \quad F_{y1} \quad F_{x2} \quad F_{y2} \cdots F_{xn} \quad F_{y_n})^T$$

where F_{xi}, F_{y_i} etc. are the components in the x and y directions, of the equivalent loads at each node of the mesh.
The following two types of loadings are considered here:

Point loads If there is a load P applied at one point of the body, we design our element mesh so that a node occurs at that point. The x and y components of the load are then added to the appropriate entries in $\vec{\mathbf{f}}$. In 2D, point loads are in fact line loads extending in the out-of-plane direction. In a plane stress analysis where the thickness is not 1.0, you should take this into account when calculating the load to apply. In an axisymmetric analysis with ELAST or ELADV, the load is multiplied by the circumference at that point.

Surface tractions Consider a pressure p applied along the $\eta = 1$ edge of an element, as in the diagram at the end of the Chapter. p may vary along the edge, ie. $p = p(\xi)$. As the local coordinate system will not be the same as the global coordinates, there will usually be both x and y components of pressure. Our task is to find the components of the equivalent nodal forces on the nodes along the loaded side: $p_{x5}, p_{y5}, p_{x6}, p_{y6}, p_{x7}, p_{y7}$. Consider a small element $d\xi$ along the loaded side. The force applied to this element is $p\, d\xi$.
The x and y components of this force are

$$p_x = p\,\frac{\partial y}{\partial \xi}\,d\xi \qquad \text{and} \qquad p_y = -p\,\frac{\partial x}{\partial \xi}\,d\xi. \tag{4.27}$$

The equivalent nodal forces are then given by

$$p_{x5} = \int_{-1}^{1} p\, N_1\, \frac{\partial y}{\partial \xi}\, d\xi$$

and

$$p_{y5} = \int_{-1}^{1} p\, N_1\, (-\frac{\partial x}{\partial \xi})\, d\xi, \tag{4.28}$$

and similarly for nodes 6 and 7, using N_2 and N_3. Here, the shape functions are the quadratic functions in one dimension:

$$N_1 = \xi(1-\xi)/2, N_2 = 1-\xi^2, N_3 = \xi(1+\xi)/2$$

These integrals can then be evaluated numerically by a two-point Gaussian rule.

4.2 Program operation

The program is compiled and run in the same way as described for the Poisson program in Section 3.2. It must also be linked with the file `FELSB.FOR`. `ELAST.FOR` operates in exactly the same way as the Poisson program, as described in Section 3.3, producing a `.prt` print file and a `.out` output file for post-processing.

When the program is run, the user is prompted to specify whether the analysis is under conditions of plane strain, plane stress or axisymmetry; this is done by typing 1, 2 or 3 respectively. Note that this is not specified in the data file. `ELADV.FOR` also handles all three conditions, but the plasticity programs are restricted to plane strain.

There are four material properties used in `ELAST`:

1. Young's Modulus E;
2. Poisson Ratio ν;
3. Thickness T (only used in plane stress analyses; otherwise leave as 0.0);

4. Tensile strength σ_{ten} (optional).

The tensile strength is only used to test for yield, in subroutine `result`. Note that no plastic flow is involved; if you need to perfrom a plasticity analysis, use the Advanced-level `PLAST.FOR` or `VPLAST.FOR` 'main engines'.

4.3 Program structure

After reading the mesh data from the input file, and echoing it to the output and print files, the main program prompts you to choose whether a plane strain or plane stress analysis is required, and then performs the finite element calculation by calling the following subroutines in turn:

`stiff` This subroutine calculates the element stiffness matrices and writes them to the scratch file.

`load` This subroutine reads the loading data from the input file, and calculates the equivalent nodal load vector f, stored in the array `aslod`.

`assmb` This subroutine assembles the element stiffness matrices to form the global stiffness matrix.

`solve` This subroutine solves the global stiffness equation (4.16) for the nodal displacements.

`result` This subroutine uses the nodal displacement results to calculate stresses at the Gauss points of each element. It then writes this displacement and stress information to the print and output files.

Some more detail on each of these subroutines is now given. Much of the coding is identical to the corresponding subroutines in the Poisson program, and the description in Section 3.4 should be read first. The matrix assembly and solution algorithms will be described in Chapter 8. Note that the source code is provided with comment lines explaining the sections, and describing the variables used. A list of the more important variables is given here, for reference.

(1) *Integer Variables*

NELEM	number of elements in mesh (max. = MELEM)
NBAND	halfbandwidth+1 of global stiffness matrix (max. = MBAND)
NNODE	number of nodes in a single element (max. = MNODE)

NPOIN	number of nodes in mesh (max. = MPOIN)
NPROS	number of material property sets
NTOTV	number of degrees of freedom (max. = MTOTV), where NTOTV = 2 * NPOIN

(2) *Real Variables*

DJACB	Jacobian determinant
G	shear modulus $\frac{E}{2(1+\nu)}$
PR	Poisson's ratio ν
YM	Young's modulus E
THICK	Material thickness T (used in plane stress analyses)
SIGT	Tensile strength σ_{ten} (used in `result`)

(3) *Integer Arrays*

LPROS(MELEM)	material property set number of a single element
NKODE(MNODE)	fixity code for each node
LNODS(MELEM,8)	global node numbers around each element

(4) *Real Arrays*

ASDIS(MTOTV)	global nodal displacement
ASLOD(MTOTV)	global nodal load vector
GPEPS(3)	Gauss point strains
GPSIG(3)	Gauss point stresses
GPWTS(4)	Gauss weights
SHAPE(8)	shape functions
BMATX(3,16)	B matrix
CARTD(2,8)	Cartesian derivatives of shape functions
DERIV(3,8)	local derivatives of shape functions
DMATX(3,3)	D matrix
ENCOD(2,8)	nodal coordinates for the node around the current element
ESTIF(16,16)	element stiffness matrix
GPLOC(2,4)	Gauss point local coordinates
GSTIF(MTOTV,MBAND)	global stiffness matrix
PRESS(2,3)	surface tractions on the three nodes
PROPS(4,2)	material property set values

4.3.1 `stiff`

Within the element loop, this subroutine first of all identifies which material property set that element is in, and then extracts its Young's modulus and Poisson's ratio from PROPS.

To deal with the double integral in (4.24) the matrix multiplications in the integrand are carried out in the `do` loops 39,40, and 49, and the integral is approximated by the 2×2 Gaussian rule (4.25) in the `do` loop 75.

The element stiffness matrix is formed by using the following subroutines:

`DMAT` creates the D matrix and returns it in DMATX;

`GAUSS` returns the Gauss point coordinates in GPLOC, and weights in GPWTS;

`SFR` creates the shape functions N_i for the isoparametric quadrilateral, evaluated at the current point, and returns them in SHAPE; these functions are

$$\mathbf{N}(\xi,\eta) = \frac{1}{4}\begin{bmatrix} (1-\xi)(1-\eta)(-\xi-\eta-1) \\ 2(1-\xi^2)(1-\eta) \\ (1+\xi)(1-\eta)(\xi-\eta-1) \\ 2(1+\xi)(1-\eta^2) \\ (1+\xi)(1+\eta)(\xi+\eta-1) \\ 2(1-\xi^2)(1+\eta) \\ (1\ -\xi)(1+\eta)(-\xi+\eta-1) \\ 2(1-\xi)(1-\eta^2) \end{bmatrix}$$

It also forms their local derivatives with respect to ξ and η, and returns these in the array DERIV.

`JACOB` creates the Jacobian matrix XJACM in (4.20) by, for example

$$\frac{\partial x}{\partial \xi} = \sum_{i=1}^{8} \frac{\partial N_i}{\partial \xi} x_i \qquad (= \sum \text{DERIV} \times \text{ENCOD})$$

where

$$\text{XJACM} = \begin{bmatrix} \dfrac{\partial x}{\partial \xi} & \dfrac{\partial y}{\partial \xi} \\ \dfrac{\partial x}{\partial \eta} & \dfrac{\partial y}{\partial \eta} \end{bmatrix}$$

and uses Cramer's rule to find the inverse of XJACM, where

$$\text{XJACI} = (\text{XJACM})^{-1}$$

It also uses XJACI to find the Cartesian derivatives of the shape functions by the formula (4.22).

`BMAT` forms the **B** matrix in the array BMATX. This array has three rows and 16 columns, made up of a submatrix

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 \\ 0 & \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \end{bmatrix}$$

for each node.

When the stiffness matrix ESTIF for each element has been formed, it is written in a temporary scratch file.

4.3.2 `load`

This subroutine reads the loading data, and forms the global load vector ASLOD. There are two types of loading that we consider: point loads and surface tractions.

For point loads, data is read in from the statement

```
15 read(15,520) lcard,ld,np,fx,fy
```

where `np` is the global node number of the node where the point load is applied, and `fx` and `fy` are the x and y components of the point load.

For surface tractions, there may be several different sets of surface traction. Details of each set are read from the statements

```
25 read(15,530) lcard,iset,i1,i2,pn,pt
```

where `iset` is the surface traction set number, and `pn` and `pt` are the normal and tangential components of the traction. These tractions are associated with loaded element edges by reading the data

```
30 read(15,540) lcard,nnas,l,n1,iset1,n3,iset3
```

where `l` is the element number, `n1` and `n2` are the corner nodes defining the loaded edge of the element, and where the traction load has the values in surface traction sets `iset1` and `iset2` respectively.

4.3.3 `assmb`

The assembly subroutine operates as in the Poisson program (i.e. storing the sub-diagonal band of the global stiffness matrix), but with the added complication that there are now two degrees of freedom (the x and y displacements) at each node. The halfbandwith is thus

```
NHBW = 2*MDIFF + 1
```

where `MDIFF` is the maximum difference in node numbers within an element, over all elements. The mapping of the entries in K into the compact storage array `gstif` is also modified, for this reason. See Chapter 8 below, for full details of the assembly algorithm. As with `POISS.FOR`, the compact storage of the band requires NBAND columns, where NBAND = NHBW + 1.

4.3.4 `solve`

This subroutine solves the matrix equation $\mathbf{K}\vec{\mathbf{u}} = \vec{\mathbf{f}}$ for $\vec{\mathbf{u}}$, where:
$\mathbf{K}$ is the global stiffness matrix GSTIF,
$\vec{\mathbf{u}}$ is the global displacement vector ASDIS, and
$\vec{\mathbf{f}}$ is the global load vector ASLOD.

The method of solution of this matrix equation is provided by Choleski decomposition. Before solving the stiffness matrix equation, the boundary conditions must be introduced according to fixity codes of the nodes on the boundary. If a nodal point is fixed in either x or y or both directions, a factor of 10^{20} is added to the diagonal coefficient of its corresponding row in the global stiffness matrix. Physically, it corresponds to 'earthing' the structure with a very stiff spring. After finding the solution we obtain a very small displacement instead of an absolute zero, and the reaction for that support can be computed directly as

$$\text{reaction} = -(\text{big spring stiffness}) \times (\text{very small displacement}).$$

This technique is discussed in Chapter 8.

To each node, a fixity code NKODE is assigned. It contains two digits; the i'th digit of the code is 0 if that component is free, and 1 if it is fixed. For example, if the fixity code of the node with global node number 8 is 01, it means that node 8 is free to move in the x direction, but fixed in the y direction.

The following subroutines are used:

`FIXDOF` introduces boundary conditions by modifying the global stiffness matrix, as described above.

`CHODEC` performs the Choleski decomposition of K as $L.L^T$, where L is a lower diagonal matrix.

`CHOSUB` uses forward and backward substitution to solve for the nodal displacements in the array `ASDIS`.

`REACTN` zeroes the load vector `aslod`, and then finds the fixed degrees of freedom, and calculates the reaction forces at those nodes. These are stored in `aslod`, to be printed out later.

Further details of the programming involved in these subroutines, is given in Chapter 8.

4.3.5 `result`

This subroutine appends the displacement and stress results to the output file. There are two blocks of results, each preceded by a one-line header, with format as shown in the following table:

Block	Format	Variables
R1	I5,2E10.3	`N,DISPX,DISPY`
R2	I5,I2,I3,5E10.3	`L,IJ,KODE,GX,GY,SX,SY,TXY`

where:

`N` = Node number
`DISPX` = Displacement in x direction at node `N`
`DISPY` = Displacement in y direction at node `N`
`L` = Element number
`IJ` = Gauss point number (1, 2, 3 or 4) within element `L`
`KODE` = Yield code for the Gauss-point (see below)
`SX` = Stress σ_x at Gauss point `IJ` of element `L`
`SY` = Stress σ_y at Gauss point `IJ` of element `L`
`TXY` = Stress τ_{xy} at Gauss point `IJ` of element `L`

The best points at which to evaluate the stresses are the four Gauss points which were used to perform numerical integration over the element. The stresses are calculated from the vector of displacements at the element's nodes, u_e, by

$$\sigma = \mathbf{DBu_e}$$

where the B matrix is evaluated at the Gauss point.

If a non-zero value has been entered as the 4th material property, this is used as the tensile strength σ_{ten} of the material. If the minor principal stress is tensile and exceeds this strength, then `KODE` is set to 1. The Gauss-points at which this has occurred, can be seen in the pre-processor using the 'Yield Zones' display option. Note that σ_{ten} should

be given as a positive number; for example, if $\sigma_{\text{ten}} = 0.1$MPa, then the Gauss point will have yielded if $\sigma_3 < -0.1$.

4.4 Solution of axisymmetric problems

When ELAST runs, the user is asked to indicate the type of analysis: type 1 for plane strain, 2 for plane stress, and 3 for an axisymmetric analysis. In this last case, the stress and strain vectors have an additional fourth component: the hoop stress/strain. The vectors are thus $\vec{\sigma} = (\sigma_r, \sigma_z, \tau_{rz}, \sigma_\theta)^T$ and $\vec{\epsilon} = (\epsilon_r, \epsilon_z, \gamma_{rz}, \epsilon_\theta)^T$. The D and B matrices are as defined in §4.1.1 for plane strain, but with an extra row/column:

$$\mathbf{D} = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & 1 & 0 & \frac{\nu}{1-\nu} \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 1 \end{bmatrix} \tag{4.29}$$

and

$$\mathbf{B} = \begin{bmatrix} \frac{\partial N_1}{\partial r} & 0 & \frac{\partial N_2}{\partial r} & 0 & \cdots & \frac{\partial N_8}{\partial r} & 0 \\ 0 & \frac{\partial N_1}{\partial z} & 0 & \frac{\partial N_2}{\partial z} & \cdots & 0 & \frac{\partial N_8}{\partial z} \\ \frac{\partial N_1}{\partial z} & \frac{\partial N_1}{\partial r} & \frac{\partial N_2}{\partial z} & \frac{\partial N_2}{\partial r} & \cdots & \frac{\partial N_8}{\partial z} & \frac{\partial N_8}{\partial r} \\ \frac{N_1}{r} & 0 & \frac{N_2}{r} & 0 & \cdots & \frac{N_8}{r} & 0 \end{bmatrix} \tag{4.30}$$

. The dimensions of the arrays DMATX, BMATX, etc. are therefore increased from those given earlier in this Chapter, to accommodate the extra rows/columns. Also, in axisymmetry the integrations must include a factor of $2\pi r$ to represent integration around the circumference; this is applied (instead of multiplying by the out-of-plane thickness) in calculating `dvolu`, when forming the stiffness matrix, and in converting surface tractions to equivalent nodal loads. It is also used to multiply any point-loads being applied. The value of r is taken from the first coordinate component of the node or Gauss point, and named `radius`.

Axisymmetric meshes with loads can be prepared in PREFEL, the user interpreting all references to x, y as corresponding to the r, z directions. The same reinterpretation should be used while viewing the results in FELVUE. When viewing stress contours, there is an option for viewing the out-of-plane stress σ_z as the fourth stress component; this is used in axisymmetry to view the hoop stress σ_θ. Note that the principal stresses will be those in the rz-plane; the hoop stress may be the major or minor principal stress in 3D.

The data file eldemaxi.dat is an axisymmetric mesh to analyse the thick cylinder problem solved in plane strain in eldemo.dat (see Chapter 9). The same displacements of the outer and inner circumferences, are obtained from the two analyses.

Chapter 5

The beam and frame program FRAME.FOR

The PREFEL pre-processor allows you to use in your mesh one-dimensional line elements containing two nodes or three nodes (at the two ends and the mid-point of the line). In elasticity analyses, these may represent either bars (which have axial stiffness) or beams (which have axial and flexural stiffness). The advanced elasticity program ELADV.FOR interprets such elements as beams, although the subroutines which evaluate the shape functions, etc. also contain the coding for the simpler bar elements.

5.1 Bar or truss elements

The theory for the elastic bar element is a one-dimensional version of the 2D elasticity theory described in Section 4. These elements are also called truss elements. The constitutive equation is

$$\sigma = \mathbf{E}\epsilon \tag{5.1}$$

where σ, ϵ are the axial stress and strain, and $\mathbf{E}$ is the Young's modulus. Similarly, the strain-displacement relation is

$$\epsilon = \frac{du}{dx}. \tag{5.2}$$

Working through the theory of Chapter 4, and remembering that integration over the element involves multiplication by the cross-sectional area A of the bar, the reader will be able to determine the stiffness matrix for the linear bar element as:

$$K_x = \frac{EA}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{5.3}$$

where L is the length of the bar. For reference, the linear shape functions in the local ξ coordinate are:

$$N_1 = \frac{1}{2}(1-\xi), \text{and } N_2 = \frac{1}{2}(1+\xi)$$

where the nodes are at $\xi = \pm 1$.

The matrix K_x relates axial forces at the nodes, to the axial displacements at the nodes; the x subscript indicates this. If the bar is oriented at an angle in the xy plane, the matrix must be transformed by a rotation matrix; this will be done later.

For a 3-noded bar element, the force-displacement matrix may be derived similarly, or be formed using Gaussian integration as with the 2D elastic elements. The shape functions, derivatives, etc. for 2-noded and 3-noded bar elements are provided in `ELADV.FOR`, should the user wish to program this.

More useful, however, are beam elements, which can support flexural as well as axial loads. The analysis of the axial load-displacement relationship is identical to that given above for bars; we will now describe the behaviour of the beam under transverse loading. The following is necessarily a brief summary, without the underlying stress theory.

5.2 Two-noded beam element

The flexural stiffness of the beam is defined by its Moment of Inertia I. The governing differential equation is

$$EI\frac{d^4v(x)}{dx^4} = w(x), \tag{5.4}$$

where x is the axial direction, $v(x)$ is the transverse displacement of the beam, and $w(x)$ is the external tranverse load applied. The slope of the beam at any point, relative to the x-axis, is given by v' and if this is denoted by θ we assume small-deflection theory in which $\theta \doteq \sin\theta \doteq \tan\theta$. In beam theory, we treat the rotation θ as a third degree of freedom, together with the displacements (u, v).

Under this theory, the bending moment $M(x)$ is related to the displacement by

$$EI\frac{d^2v(x)}{dx^2} = M(x), \tag{5.5}$$

and

$$\sigma(x) = \frac{M(x)\ y}{I}$$

is the stress across the beam caused by a bending moment M; y is the axis normal to x. The variational formulation involves minimizing the potential energy$J(v)$ where

$$J(v) = \int_A \int_0^L \frac{1}{2}\sigma\epsilon\ dx\ dA - \int_0^L w\ v\ dx \tag{5.6}$$

where the beam extends from $x = 0$ to $x = L$. Then

$$\epsilon(x) = \frac{1}{E}\sigma(x) = \frac{1}{EI}M(x)\ y = y\frac{d^2v}{dx^2}.$$

Substituting in (5.6),

$$J(v) = \int_A \int_0^L \frac{1}{2}y^2(\frac{d^2v}{dx^2})^2 dx\ dA - \int_0^L w\ v\ dx. \tag{5.7}$$

But from the definition of Moment of Inertia,

$$\int_A y^2 dA = I,$$

so

$$J(v) = \frac{1}{2}\int_0^L EI(v'')^2 dx - \int_0^L w\ v\ dx. \tag{5.8}$$

5.2.1 Element stiffness matrix

We now introduce the shape functions. The transverse displacement will be a linear combination of the nodal displacements and rotations. We will work with a two-noded beam element, with end-nodes 1 and 2:

$$v(x) = N_1(x)\ v_1 + N_2(x)\ \theta_1 + N_3(x)\ v_2 + N_4(x)\ \theta_2 = [N]\ \vec{v}. \tag{5.9}$$

As $v'' = [N'']\vec{v}$, we substitute in (5.8) to get

$$J(v) = \frac{1}{2}\int_0^L \vec{v}^T\ [N'']^T EI[N'']\ \vec{v}\ dx - \int_0^L \vec{v}^T[N]^T\ w\ dx. \tag{5.10}$$

Minimize $J(v)$ with respect to $\vec{v}$ and set each row to zero to produce

$$Kv = F \tag{5.11}$$

where K is formed from assembled element stiffness matrices

$$K_e = \int_0^L [N'']^T EI[N'']\vec{v}\ dx$$

and the load vector is assembled from

$$F_e == \int_0^L [N]^T w\ dx.$$

The shape functions $N_i(x)$ must satisfy the following conditions:

$$N_1(0) = N_3(L) = 1 \tag{5.12}$$
$$N_1(L) = N_3(0) = 0 \tag{5.13}$$
$$N_2(0) = N_4(0) = N_2(L) = N_4(L) = 0 \tag{5.14}$$
$$N_1'(0) = N_3'(0) = N_1'(L) = N_3'(L) = 0 \tag{5.15}$$
$$N_2'(0) = N_4'(L) = 1 \tag{5.16}$$
$$N_2'(L) = N_4'(L) = 0 \tag{5.17}$$

which ensures that $v(0) = v_1, v(L) = v_2$ and that $\frac{dv}{dx} = \theta$ at the two nodes. These are satisfied by the Hermite cubic shape functions:

$$N_1 = 1 - 3\frac{x^2}{L^2} + 2\frac{x^3}{L^3} \tag{5.18}$$

$$N_2 = x(1 - 2\frac{x}{L} + \frac{x^2}{L^2}) \tag{5.19}$$

$$N_3 = 3\frac{x^2}{L^2} - 2\frac{x^3}{L^3} \tag{5.20}$$

$$N_4 = x(\frac{x^2}{L^2} - \frac{x}{L}) \tag{5.21}$$

$$\tag{5.22}$$

Hence, the element stiffness matrix K, whose degrees of freedom correspond to the vector of element nodal variables $(v_1, \theta_1, v_2, \theta_2)^T$, is:

$$K_{xy} = \frac{EI}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix} \tag{5.23}$$

This matrix for the (v, θ) degrees of freedom must be combined with the stiffness matrix (5.3) derived earlier for the u (axial) degree of freedom, to produce a 6×6 stiffness matrix for a beam with both axial and flexural stiffness. This matrix is for the xy coordinate system oriented with the beam; if the beam is inclined at an angle α to the true x-axis in our Cartesian coordinates, it must be transformed $T^T K T$ where the rotation matrix contains submatrices

$$\begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.24}$$

The resulting element matrix is programmed explicitly in subroutine `estif` in program FRAME.FOR.

5.2.2 Equivalent nodal load vector

The equivalent nodal load vector is obtained by evaluating the right-hand-side of (5.11). First of all, point loads applied at nodes are dealt with in the same way as in `ELAST.FOR`; we now consider applied loads $w(x)$ distributed along the length of the beam element. In integrating $N^T w$, the load vector $w(x)$ contains the distributed loads $w_a(x)$ in the axial and $w_n(x)$ in the normal directions, and applied distributed moments. In `FRAME.FOR` we do not consider applied moments, as these are not often required in practice. The shape functions in N are those derived in the previous section. We allow the surface tractions to vary linearly along the beam, from a value of $w_1(x)$ at node 1 to $w_2(x)$ at node 2.

Then, if the integrations are performed, the following expressions are obtained:

- Tractions in the axial directions use the simple linear shape functions:

$$N_1 = 1 - x/L, N_2 = x/L,$$

  and the equivalent nodal loads are $(2w_{a1} + w_{a2})L/6$ at node 1, and $(w_{a1} + 2w_{a2})L/6$ at node 2.

- Tractions normal to the axis of the beam use the cubic shape functions in (5.18), and it is found that these give rise to nodal moments as well as nodal normal loads. The nodal normal loads are $(7w_{n1} + 3w_{n2})L/20$ at node 1, and $(3w_{n1} + 7w_{n2})L/20$ at node 2; there are also applied nodal moments of $-(3w_{n1} + 2w_{n2})L^2/60$ at node 1 and $(2w_{n1} + 3w_{n2})L^2/60$ at node 2.

To understand the sign convention relating to distributed loads, you should imagine that the beam element comprises the first side of a two-dimensional element, with the nodes numbered anti-clockwise around the element. Then, normal loads applied in a direction compressing the 2D element, are given positive sign.

The above explicit formulas for equivalent nodal loads/moments are programmed into subroutine `load` in `FRAME.FOR`.

5.2.3 Program structure

Apart from the explicit formulations for the element stiffness matrices and equivalent load vector, described in the last two sections, the programe structure is otherwise identical to that in `ELAST.FOR`, except that there is no equivalent for beams to the Gauss-point stresses post-calculated in `result`.

There is also no equivalent to the subroutine `assmb`. This is because in `FRAME.FOR` an iterative solver has been used for solving the global matrix equation (5.11). This is the conjugate gradient method with diagonal preconditioning. It is described in Chapter 8 on global matrix storage and equation solution algorithms. For this solver, there is no need to assemble the global stiffness matrix K; this matrix is only used in forming matrix-vector multiplications Kv. This calculation in performed in subroutine `ematgv` by reading each element matrix in turn from the scratch file where they were written by `stiff`, multiplying each entry by the appropriate entry in v and adding to the appropriate entry in the output vector.

Sample datafiles `framexa1.dat, framexb1.dat` are provided; these are documented in Chapter 9.

5.3 Three-noded beam elements

Engineers often desire to use beam elements in the same mesh as 2D elastic elements – in problems involving soil-structure interaction, for example. If a two-noded beam element is joined along an edge in a mesh of 8-noded quadrilateral elements – to represent a beam buried in soil, for example – an unrealistic stress distribution results, because forces are only transmitted between beam and soil at the beam nodes. A better, though not perfect, model can be obtained if a 3-noded beam element is used. We now describe such an element, where the (u, v, θ) degrees of freedom exist at the two end nodes, and at the mid-node there is only (u, v) displacement degrees of freedom. This is analogous to the composite element used for soil consolidation in program `CONSL.FOR` (where the 3rd d.o.f. represents pore pressure). Such an element was derived in the PhD thesis of Chao (see Chapter 10).

In this element, we introduce a local coordinate ξ, where nodes 1,2,3 are at $\xi = -1, 0, 1$ respectively, so that $x = (1 + \xi)L/2$. Hence $d\xi/dx = 2/L$.

The shape functions relating to the axial displacement are precisely the 1D quadratic functions used for surface tractions in subroutine `load` in `ELAST.FOR`, namely:

$$N_1 = \xi(1 + \xi)/2, N_2 = 1 - \xi^2, N_3 = \xi(1 - \xi)/2.$$

There will be five shape functions relating to normal displacement, which is formed as

$$v(x) = N_1(\xi)v_1 + N_2(\xi)v_2 + N_3(\xi)v_3 + N_4(\xi)\theta_1 + N_5(\xi)\theta_3. \tag{5.25}$$

If we assume $N_1, \ldots, N_5$ to be arbitrary quartics, and impose the conditions that $v(x_1) = v_1, v'(x_1) = \theta_1$ etc., we find that

$$\begin{aligned}
N_1 &= \xi(-3 + 4\xi + \xi^2 - 2\xi^3)/4 && (5.26)\\
N_2 &= 1 - 2\xi^2 + \xi^4 && (5.27)\\
N_3 &= \xi(3 + 4\xi - \xi^2 - 2\xi^3)/4 && (5.28)\\
N_4 &= \xi(-1 + \xi + \xi^2 - \xi^3)L/4 && (5.29)\\
N_5 &= \xi(-1 - \xi + \xi^2 + \xi^3)L/4 && (5.30)\\
& && (5.31)
\end{aligned}$$

Explicit formulas for the stiffness matrix entries, and for converting distributed loads to equivalent nodal loads can then be derived following the process of the previous section.

This element has not been programmed into `FRAME.FOR`, but it has been included in the advanced elasticity program `ELADV.FOR`; both 2-noded and 3-noded beam elements are included there, and the explicit form of the element stiffness matrices, after the rotation transformation has been applied, are given in subroutine `estifb`. Subroutine

`load` shows the formulas for equivalent nodal loads (The 2-noded and 3-noded beam elements are identified by `NODSID=21` and `NODSID=31` respectively). There is a datafile `eladbm1.dat` which models a beam part-embedded in an elastic block, with a normal surface traction applied on the element at its tip; see Chapter 9 for more details.

Chapter 6

Using the post-processor

The post-processor, FELVUE, is started by clicking on its icon.

Within the post-processor, you will be able to change directory to your user directory, and then a list of output files with `.out` filename extensions will be displayed. Choose an output file, and the post-processor will first read the mesh data, and display this as in the pre-processor. Click on *Continue*, to proceed to a display of the results.

Note that the 'main engines' output the results (displacements, stresses, potentials, etc.) in E10.3 format, so that very large numbers can be handled, and reasonable accuracy obtained with very small ones. However, if you know that all your stresses will be of a certain order of magnitude such as 10-100, you may wish to edit the format statement to output then in F10.5 format, say. FELVUE reads real numbers from the data-file in G10.3 format, so that it will accept numbers to 5 decimal places, and these will then be output to this accuracy in the pop-up windows described below.

It is possible to produce plots which only show elements of a specified material type (LMTYPE). If you want all elements plotted, leave LMTYPE=0 in the dialogue box which offers this facility. The following sections now describe the plots available from the Main Menu.

All references to displacements, stresses etc. in the xy-plane should be reinterpreted as meaning the rz-plane in axisymmetry. Note that an out-of-plane stress can also be plotted, as the fourth stress component; this is the hoop stress in axisymmetry.

6.1 Scalar-variable meshes

In the case of a scalar-variable mesh, the data read from the .OUT output file comprises:

- nodal values of the scalar potential (e.g. nodal temperatures);

- flow rates in x and y directions at element Gauss points.

The choice of plots (all of which can be zoomed) is:

6.1.1 Yield zones

These are described in the elasticity-plot section below.

6.1.2 Flow contours

The horizontal and vertical flow rates q_x, q_y can be plotted as contours, either using coloured contour lines or with colour fill-in. You can plot either horizontal or vertical flow components, or plot contours of the flow magnitude $\sqrt{q_x^2 + q_y^2}$. Note that if the mesh is of linear triangles, there is only one Gauss point per element, so the plot will colour the whole of each element in the appropriate colour; the 'lines'-type plot will not work in this case.

6.1.3 Temperatures

This is the main plot, of nodal potentials U. A contour plot of the potential field, interpolated from the nodal values, will be displayed. When a node is picked using the mouse, the nodal coordinates and nodal value of U will be displayed in the dialogue box.

6.1.4 Flowlines

This is a more useful form of plot of the flow-rates q_x, q_y, as flow-lines at the Gauss-points, showing the direction and magnitude of flow, the latter relative to a reference value which the user can enter (default = 100.0). Decreasing the reference value, increases the lengths of the flowlines. With the 'Pick Gauss point' menu option, the user can select a Gauss-point by clicking on it, and a dialogue box will advise the flow-rates at that point.

6.2 Vector-variable meshes

For elasticity-type problems, the data read in is:

- nodal values of the x and y displacements;

- stresses at element Gauss points.

The choice of plots (all of which can be zoomed) is:

6.2.1 Displacements

There are three ways in which the displacement field (u, v) can be viewed:

Deformed mesh

The mesh is drawn with the nodes displaced from their original coordinates, by the displacements solved for in `asdis`, multiplied by an exaggeration factor. The initial exaggeration factor is calculated so as to achieve a 10you can then re-draw with a greater or smaller exaggeration as required. If you pick a node, its displacement components will be reported in the pop-up window.

Displacement contours

This is a contour plot of the displacement magnitude $\sqrt{u^2 + v^2}$

Nodal vectors

This shows the nodal displacements (u, v) as vectors from the node positions. This is a more meaningful plot for analyses where u, v do not represent displacements but velocities, etc. By picking a node, the values of u, v will be displayed.

6.2.2 Plot 1D elements

If the post-processor detects any one-dimensional line elements in the mesh, it will ask if these elements represent beams. If you answer 'y', the beam element shape functions derived in Chapter 5, together with the nodal displacements and rotations, will be used in plotting these elements. This is the case both for the standard 2-noded cubic beam elements, and the higher-order 3-noded quartic beam element described in Chapter 5. If you answer 'n', the standard 1D linear or quadratic shape functions will be used with the nodal displacements; this should be used when the elements represent bars or trusses; this should also be used if the isoparametric beam element described in the text by Hinton and Owen (see Chapter 10) has been programmed.

The 'Plot 1D elts' option will plot the deformed 1D elements (bars, beams and joints) together with a deformed boundary of any 2D mesh; this can be seen in the example datafile `eladbm1.dat`, of a beam part-embedded in an elastic block. The 1D elements are not plotted in the 'Displacements' option, so as not to complicate the display.

6.2.3 Stresses

Stresses at Gauss points can be displayed as stress crosses, or extrapolated to a stress field which is contoured element-by-element, for a chosen stress component. Remember that FELIPE uses the compression-positive sign convention, so that the major principal stress will be the most positive (or least negative) in this sense. If both principal stresses are tensile at a particular point, the major principal stress will be the smaller tensile stress. In addition to the in-plane stresses, you can plot a fourth stress component, the out-of-plane stress σ_z (plane stress/strain) or σ_θ (axisymmetry).

Stress crosses

The stress tensors at the Gauss points are resolved into the principal stresses, and a 'stress cross' drawn at each point. The lengths of the arms of the stress cross indicate the magnitudes of the major and minor stresses (in proportion to the reference stress which you must type in), and their directions indicate the principal stress directions. Compressive stresses are drawn in black, and tensile stresses in blue. The initial scaling factor is set to give a reasonable-size stress cross for the average stress in the mesh, but where there are areas of high concentration this will result in very large stress crosses at those points; you should try another, smaller scaling factor in this case.

Stress contours

Contours of stress are also drawn, interpolated within each element from the Gauss point stress values. You can contour major principal stress σ_1, minor principal stress σ_3, the stress average $(\sigma_1 + \sigma_3)/2$ or the deviator stress $(\sigma_1 - \sigma_3)/2$, as well as for each individual stress component. The contours can be filled-in in colour, or drawn as coloured lines (which is the option used for PostScript printing). The scales are written along the bottom. Note that the contouring is performed element-by-element, based on the element Gauss point values; thus, there will be discontinuities in stress at the inter-element edges. In a well-designed mesh, these will be small.

6.2.4 Yield zones

In Advanced-level analyses, the main program may test the Gauss point stresses against a yield criterion, such as the Mohr-Coulomb criterion, and use some nonlinear constitutive algorithm such as plasticity where yield occurs. To indicate Gauss points where yielding has occurred, a 'yield code' may be written in the results for the stresses (see format list below). This is used in the plasticity 'main engines' `PLAST, PLADV, VPLAS`, but the elasticity programs `ELAST, ELADV` can also test the minor principal stress for yield against the tensile strength.

Under this option, you enter a yield code, and Gauss points with that code will be highlighted. A yield code of 0 means the Gauss point has not yielded. The FELIPE 'main engines' use a yield code of 1, but if you have developed a more complex plasticity program and wish to distinguish different types of yield (e.g. where several different yield criteria operate independently) you can assign a different code to each type.

6.3 Plots of nodal degrees of freedom

Following the plots described above, the user is offered the chance to view contour plots of each nodal degree of freedom. This is to allow for extra degrees of freedom (temperatures, pore pressures, etc) existing, though contours of displacements in the x-direction can also be obtained here, for example. If the extra d.o.f. exists only at corner nodes (as with the poroelasticity application `CONSL.FOR`), the contouring will take account of this when specified by the user.

The output data format read by FELVUE, including such extra d.o.f.s, is:

Block	Format	Variables
R1	I5,4G10.3	`N,DISPX,DISPY,disp3,disp4`
R2	I5,I2,**I3**,6G10.3	`L,IJ,`**KODE**`,GX,GY,SX,SY,TXY,sz`

where **KODE** is an integer yield code (Non-yielded points take code 0). This is used in PLAST and VPLAS, and is also available in ELAST and ELADV to test if the tensile strength has been exceeded; in these programs, yielded points are indicated by KODE=1. Of course, you can develop your own 'main engines' and use other values of KODE to indicate other types of yielding, or indeed you can use this facility to distinguish other types of behaviour besides yielding.

Note that the G10.3 format is used to read real values. The 'main engines provided will output real values of displacements, stresses, etc. to the .out file in E10.3 format, so that very large and very small numbers can be coped with. However, greater accuracy for values of a particular order of magnitude can be achieved by editing the 'main engine source code to write in, for example, F10.5 format. This will be read equally happily by FELVUE.

Following the examination of d.o.f. plots, the user has the option of returning to the Main Menu if desired.

6.4 Postscript file plots

When you have finished inspecting the results, the screen reverts to Text mode, and you have the option to produce PostScript files. If the output file were called `test.out`, then the PostScript file of the deformed mesh would be called `testa.ps`, a PostScript file of the stress crosses would be `testb.ps`, a PostScript file of the stress contours would be `testc.ps`, and a PostScript file of the yield zone would be `testd.ps`. These plots can also be zoomed, by specifying a 'window' to plot. This is done by typing in the x coordinates of the left and right sides of the window, and the y coordinates of the top and bottom of the window. If you want to use this facility, therefore, you should note the coordinates you want to use, while viewing the graphical results on screen earlier.

6.5 Plotting further result sets

It is possible to make a 'main engine' output further sets of results, separated by a title line. This is useful in a time-stepping or incremental analysis, where results after certain times or load increments may be of interest. This is used in the soil consolidation 'main engine' `CONSL.FOR`, where the initial, undrained result is output, and then the final, drained solution after completing the timestepping is appended to the `.out` file (see `conslex3.dat`).

When the post-processor detects that more datalines exist following a set of results, a pop-up box asks if you wish to view the next result set. Answering 'y' will return you to the main menu, with the next set of results.

Chapter 7

Advanced-level variants of ELAST

In addition to the POISS.FOR and ELAST.FOR 'main engines', the FELIPE package provides source code for four further 'main engines' that expand the facilities of ELAST.FOR for continuum mechanics problems, exploiting the extra features of the pre-processor available at Advanced level. These are:

ELADV.FOR is a 'big' version of ELAST, with coding for the full range of elements which can be created in the pre-processor. It also copes with the range of loading options available in PREFEL .

THERM.FOR extends ELAST to a coupled analysis of elasticity and temperature diffusion, coupled via a coefficient of thermal expansion. This demonstrates the feature in PREFEL which allows the user to create additional degrees of freedom at nodes in the mesh.

CONSL.FOR extends ELAST to the analysis of soil consolidation or poroelasticity, i.e. coupled elastic deformation and fluid flow, in this case also involving a timestepping régime.

PLAST.FOR extends ELAST to a nonlinear material model, namely elastoplasticity. Loading is added incrementally, and within each increment an iteration in the main program seeks to reduce the residual stresses according to a Mohr-Coulomb yield criterion and flow rule.

PLADV.FOR is identical to PLAST, but demonstrates a range of different matrix storage and solution techniques, allowing the user to compare performance.

VPLAS.FOR is another nonlinear version of ELAST, this time using elasto-viscoplasticity theory. This involves not only a load increment loop, but also use of another timestepping régime which can be prescribed in PREFEL .

The main features of each program, including the solution algorithm(s) used, are summarised in the Introduction chapter of this manual.

All the above programs, with one exception, need to be linked to the file FELSB.FOR containing input/output and other common subroutines. The exception is ELADV.FOR, which is completely self-standing. These programs should all be compiled in double precision (use the /dreal option in the Salford compiler). Remeber to compile `FELSB.FOR` with this option also – unpredictable errors at run-time will be generated if a 'main engine' compiled in double precision calls a subroutine in FELSB.FOR which is compiled in single precision, as the double-precision arguments of the call will not be mapped properly onto the subroutine's arguments.

The Appendix to the full manual summarises the finite element theory used in each program; a full description is beyond the scope of the manual, and the user is referred to the standard texts such as Zienkiewicz and Taylor (see Chapter 10). In this chapter some programming and data input aspects of the 'main engines' will be discussed. Note that the different algorithms for global matrix storage and solution, are described in the following Chapter and will not be covered below.

7.1 ELADV.FOR: full-scale elasticity analyses

The ELADV program is suitable for full-scale analyses, with up to 800 elements, 2000 nodes. The full range of 2D elements may be used, namely linear and quadratic triangles and quadrilaterals, the 6-noded quadrilateral, and the infinite side and corner elements. Each element has its own shape functions and Gauss integration rule, and the coding for these is included in subroutines `sfr` and `gauss`. An additional subroutine `mapfun` provides mapping functions needed in place of shape functions by the infinite elements in some stages. The subroutines `jacob` and `bmat` are also generalized. To identify the element type, the parameters `nnode` and `nside` defining no. of nodes/sides, are passed into the subroutines, and in general the DO loops which in ELAST ran from 1 to 8, now run from 1 to `nnode`, etc. Subroutines `sfr` and `gauss` also contain coding for shape functions and Gauss rules for the remaining elements available in PREFEL , namely linear and quadratic 1D truss elements, and finite and infinite joints. Users wishing to develop main programs using joint elements, should consult the specialist literature for the various models.

For further generality, the coding in ELADV does not assume two degrees of freedom at each node, but reads the no. of d.o.f.s at node n from array ndof(npoin). To relate each d.o.f. to the global d.o.f., it constructs an array nvar(npoin); the i'th d.o.f. at node n will correspond to global component nvar(n)+i. Note that this is not strictly needed in ELADV (though see next paragraph), but is useful for developing programs for coupled analyses such as CONSL.FOR.

ELADV also contains coding for one-dimensional line elements. Linear and quadratic (2-noded and 3-noded) bar or truss elements can be analysed in exactly the same way as 2D elements, and the subroutines `sfr, jacob` contain the coding for such elements

(distinguished by `NODSID = 21, 31` respectively). This coding is not however used, since such element types are in fact identified in `stiff` and `load` as cubic and quartic beam elements. The theory of these elements has beend described in Chapter 5. Their stiffness matrces are formed explicitly, not by numerical integration, and the coding for this is in subroutine `estifb`, called from `stiff`. The equivalent nodal loads for distributed loading on these elements, is also coded explicitly in `load`. Note that the end-nodes of these elements have displacement and rotation degrees of freedom, so that `ndof = 3`.

For large-scale analyses, the Gaussian elimination solver `solve` is impractical because it requires assembly of the global stiffness materix. Even with the band storage used in `assmb` this can take up too much storage. ELADV therefore replaces the assembly and solve subroutines with subroutine `front`, which uses the frontal solution algorithm of Irons. This is a direct solver which essentially performs the assembly and elimination processes simultaneously, only holding a so-called 'front' of variables currently active, in storage. The amount of storage needed depends on the maximum frontwidth. The parameters MFRON and MSTIF replace MHBW at the start of ELADV, as global storage dimensions. The vector GSTIF of dimension MSTIF holds the part of the global matrix within the current front (of max. dimension MFRON), and so MSTIF should be set to MFRON*(MFRON+1)/2 for a symmetric matrix. The subroutine writes information to scratch files on channels 19, 20 and 21 as it proceeds through the forward decomposition, reading them back when performing back-substitution. If a re-solution with a new right-hand-side vector is required, much of the former process is unnecessary, so a parameter IRSOL is set to 2 if a re-solution using an already-decomposed matrix is to be performed (e.g. with incremental loading); IRSOL=0 for the first solution, as in ELADV. To avoid continually writing to the scratch files, FRONT has a buffering feature. The size of the buffer is given by the parameter MBUFA, set to 100 in ELADV; the user can adjust this for maximum efficiency. Note that FRONT uses the array nvar for global d.o.f. positions, so is not restricted to meshes with 2 d.o.f.s per node. This automatically copes with the beam elements, which have 3 degrees of freedom at their end-nodes.

FRONT reads two vectors of specified d.o.f.s (NFVAR lists the global d.o.f.s which are fixed, and SPDIS lists the corresponding values) rather than the fixity codes in NKODE, so this information is converted from NKODE in the main program after `input`. In `load`, any specified nodal displacements are read, and this information is added to NFVAr and SPDIS; the variable NSDIS tells the total number of such d.o.f.s. A new parameter MFIXV sets the maximum number of fixed and specified d.o.f.s, dimensioning NFVAR and SPDIS.

Body force loading is handled in `load`, if NBSET>0, being converted to equivalent nodal forces as with surface tractions.

As described in the pre-processor chapter, an in situ stress field can be prescribed. This is useful in geomechanics applications, where the soil/rock mass is in a state of stress before loading. The type of stress field is decided by ISTYP, read from the first dataline in this section. The two parameters defining the stress field are also read from

this line, and are:

`ISTYP` = Stress field type (1 or 2)
`ISET` = Set no.
`REAL1` = horizontal stress σ_x (ISTYP=1) or soil density γ (ISTYP=2)
`REAL2` = vertical stress σ_y (ISTYP=1) or lateral stress ratio K_0 (ISTYP=2)

Thus, for ISTYP=1 the stress at all Gauss points of soil elements (with material type >1) is (σ_x, σ_y). For ISTYP=2, the stress state at a soil element Gauss point with coordinates $(x, y), y < 0$ is $(-K_0\gamma y, -\gamma y)$. Note that in ELADV, material type LMTYPE=1 is assumed to mean the element is a structural element with no initial stresses; soil/rock elements have LMTYPE=2,3,.... Also, with an overburden stress field, the plane $y = 0$ is taken as the ground level; no in situ stresses are imposed for elements lying above this level.

The in situ stress field at element Gauss points is stored in the array STRSG (see below), and integrated to find the equivalent nodal loads which are stored in TLOAD. This, together with zero displacements, forms the initial pre-stressed equilibrium state. If an excavation boundary is defined, equal and opposite forces normal to the boundary (representing unloading of the boundary) are read from TLOAD and added to ASLOD, the vector of nodal forces to be applied. This 'unloading of prestressed rock mass' approach is much more realistic than trying to model displacements around an excavation by 'turning on the gravity' in the rock, and can be extended to handle nonlinear geometric or constitutive models.

On the other hand, to model the construction of an embankment on a soil half-space $y \leq 0$, the in situ stress field is prescribed without giving any excavation boundaries. Gravity loading is then imposed on the embankment elements lying above $y = 0$; the body force magnitude is the soil density multiplied by the acceleration due to gravity, acting in the $-y$ direction. It would be straightforward to adapt the source code of ELADV.FOR to cope with more complicated geometries.

Stresses at Gauss points are stored in an array STRSG(3,4,MELEM); the i'th stress component of the ij'th Gauss point of element l is stored in STRSG(i,ij,l). In `result`, the stress increments are added to any initial stress state to give the final stress state, for output. ELADV has, like ELAST, the facility for testing the minor principal stress for yield, against a tensile strength input as the 4th material property. This is described in Chapter 4, and the yield zones can be viewed in the post-processor.

7.2 THERM.FOR: a coupled deformation/temperature analysis

This program analyses plane thermoelasticity problems, in which we have the temperature field $\theta(x, y)$ as an additional, scalar degree of freedom. The theory for this is summarised in the Appendix, and results in the following augmented stiffness matrix equation:

$$\begin{bmatrix} K & Q \\ 0 & -H \end{bmatrix} \begin{bmatrix} u \\ t \end{bmatrix} = \begin{bmatrix} f \\ r \end{bmatrix}$$

.

Here, K, u, f are the elastic global stiffness matrix, nodal displacement and equivalent nodal load vectors, as described in Chapter 4; t is the temperature. As in ELAST, both plane stress and plane strain is allowed for, and decided by the user at run-time.

The matrix H results from the Poisson equation for temperature diffusion:

$$k_x \frac{\partial^2 \theta}{\partial x^2} + k_y \frac{\partial^2 \theta}{\partial y^2} = -r \tag{7.1}$$

which must be discretized over the mesh. The coefficients of thermal conductivity in the x and y directions are k_x, k_y; in THERM.FOR we assume isotropy: $k_x = k_y = k$. This is exactly equivalent to the Poisson equation theory described in Chapter 3, and the form of H is given by equation (3.9). We do not consider radiating boundaries in this program; the temperature boundary conditions are either insulated (no normal flow, which is the natural boundary condition and requires no special coding) or a Dirichlet boundary on which the temperature is fixed. This is specified by fixing the temperature d.o.f.s for these boundary nodes when preparing the mesh, and then inputting the Dirichlet boundary plane data as described in Section 2.6.

The coupling of the two fields (displacement and temperature) occurs through the constitutive equation

$$\vec{\sigma} = D\vec{\epsilon} - \beta \vec{m} \theta \tag{7.2}$$

where $\vec{m} = (1, 1, 0)^T$, and β is the coefficient of thermal expansion. This is inserted in the equilibrium equation

$$B^T \vec{\sigma} + \vec{f} = 0 \tag{7.3}$$

and introduces a coupling matrix Q. The form of Q is

$$Q = \int N^T M^T B dV \tag{7.4}$$

where

$$M = \begin{bmatrix} \beta & 0 \\ 0 & \beta \\ 0 & 0 \end{bmatrix}$$

In practice, the temperature degree of freedom is inserted after the displacement degrees of freedom, for each node. We use the arrays `ndof, nvar` to relate the nodal and global degrees of freedom, as described in ELADV above. THERM has coding for two types of element: the linear 4-noded quadrilateral element (`NODSID=44`) and the 8-noded serendipity quadrilateral element (`NODSID=84`). In both cases we have temperature degrees of freedom at all nodes, so `ndof(np)=3` for each node `np` in the mesh.

Subroutine `stiff` works element-by-element, and should be compared with the corresponding subroutine in ELAST. It first notes the material properties:

1. Young's modulus E
2. Poisson ratio ν
3. Thickness T (default = 1.0)
4. Thermal conductivity k
5. Coefficient of thermal expansion β

and stores these as `ym,pr,thick,conduc,expand` respectively.
The elastic stiffness matrix is then formed in `estif(nevab,nevab)`, just as was done in ELAST (with the generalisation that the variable `nnode` (which may be 4 or 8) is used in loops instead of assuming 8). The other two matrices Q, H are then formed in `qmatx, hmatx`, according to the formulae above. Once the process has been performed at each Gauss-point, and added to perform the gaussian integration, the augmented element matrix is assembled from these matrices, in `xmatx`. The rows and columns of this array are then shuffled, to place the row/column corresponding to the temperature d.o.f. for node n, immediately after the rows/columns for the displacement d.o.f.s for that node. It is `xmatx` which is written to the scratch-file on channel 18.

Subroutine `load` is just as in ELAST, with the generalisation that surface tractions over edges of linear elements use the linear 1D shape functions to calculate equivalent nodal loads.

Subroutine `bndry` is lifted from POISS.FOR, and reads the Dirichlet boundary plane data for the 3rd (temperature) nodal degree of freedom.

Subroutine `assmb` is very similar to that in ELAST, and uses the same mapping of the band of K into the array `gstif`, namely:

$$K_{i,j} \rightarrow \text{gstif(i,j-i+nhbw+1)}$$

(where K is now the augmented element matrix, including the temperature rows/columns) with one important difference: as the element stiffness matrices are non-symmetric, the whole band – above as well as below the diagonal – must be stored. The array `gstif` thus has `nbw` columns, where `nbw = 2*nhbw+1`.

Subroutine `solve` solves the linear system. As this is non-symmetric, we cannot use the symmetric Choleski decomposition. Instead, we use the familiar Gaussian elimination algorithm. The subroutines called in succession are:

`fixdof` to add a large value to the diagonal coefficient for fixed degrees of freedom. Temperature d.o.f.s on Dirichlet boundaries are dealt with as in POISS.FOR;

`reduce` to perform the reduction of K to upper triangular form, with simultaneous elementary row operations on the r.h.s. vector `aslod`;

`backsu` to perform the back-substitution, the solution appearing in `asdis`;

`reactn` to back-calculate the reactions at fixed displacement d.o.f.s.

The results are then output: nodal displacements and temperatures, and Gauss-point stresses and strains.

The example datafiles for use with THERM, described in Chapter 9, model a cooling fin joined to a hot boiler at its base. From these results it becomes plain that the element formulations used here are prone to instability. This is particularly evident with the linear quadrialteral element, used in the file `thermln2.out`, where the famous 'hourglass' instability occurs. The mesh of quadratic elements also exhibits some instability of results, albeit much less. The solution to this problem, is to use an element formulation in which the extra degree of freedom – temperature, in this case – is given a bilinear discretization, which the displacements are modelled using the 8-noded serendipity formulation. The resulting 8-noded quadrialteral element would thus have (u, v, θ) degrees of freedom at its four corner nodes, and (u, v) degrees of freedom at the midside nodes. The programming of such an element is demonstrated in the soil consolidation program CONSL.FOR, described next.

7.3 CONSL.FOR: a coupled time-dependent deformation/flow analysis

This program is designed to model soil-structure interaction, with an elastic structure (elements with `lmtyp(l)=1`) in contact with a saturated soil/rock mass (elements with

`lmtyp(l)=2`). The program performs a time-dependent analysis, starting with an initial solution for the undrained deformations immediately the load is applied, and then stepping forward in time. The fluid flows through the soil mass, away from the loaded area, and further elastic deformation occurs. After a sufficiently long time there is no further appreciable deformation/flow, and the drained solution has been reached.

The CONSL program uses the eight-noded quadrilateral element from ELAST, but now there are additional degrees of freedom for the pore-pressures at the corner nodes of LMTYP=2 elements. The material properties for soil elements are:

1. Young's modulus E
2. Poisson ratio ν
3. Horizontal permeability k_x
4. Vertical permeability k_y
5. Effective porosity η/K_f, where K_f is the fluid bulk modulus.

The program is restricted to plane strain (as plane stress makes little sense in this application), so the material thickness defaults to 1.0.

Boundary conditions for displacements are handled using nodal fixities, as in ELAST. This is also used for the pore pressures. Note that p is properly the **excess** pore pressure, so we do not need to input an initial pore pressure distribution, and at permeable boundaries we have $p = 0$; the natural boundary condition is for an impermeable boundary.

There are now four blocks within the augmented element stiffness matrices (see the theory in the Appendix), as was the case with THERM.FOR. All the coding is generalised to allow for variable degrees of freedom per node; the array `nvar` described in ELADV above is used. In particular, the `assmb` and `solve` subroutines are generalised to use `nvar`.

The theory of elastic consolidation is summarized in the Appendix. The only difference between the notation there and that implemented in CONSL, is that the time discretization parameter θ is used in CONSL with the sense

$$u(t) \doteq (1-\theta)\ u(t_0) + \theta\ u(t_1) \tag{7.5}$$

where $t_1 = t_0 + \Delta t$. Thus, *theta* should be replaced by $1-\theta$ in the theory in the Appendix.

Suppose that we have the nodal displacements and pore pressures at time t_0 stored in the vector u_0, and we wish to calculate the displacements and pore pressures at time $t_1 = t_0 + \Delta t$, in vector u_1. The theory tells us that:

$$(\bar{X} + Y)u_1 = \bar{\theta} f_n + f_{n+1} - (\bar{\theta}\bar{X} - Y)u_0 \tag{7.6}$$

where $\bar{\theta} = (1-\theta)/\theta$ and

```
      /          T  \  Form K in estif(16,16)
      |  K    -Q   |  Form Q in qmatx(4,16)
X =   |            |  Form H in hmatx(4,4)
      |  0    -H   |  Assemble whole matrix in xmatx(20,20)
      \            /  Write xmatx to file


      /            \
      |  0     0   |  Form Q in qmatx(4,16)
Y =   |            |  Form S in smatx(4,4)
      | -Q    -S   |  Assemble whole matrix in ymatx(20,20)
      \            /  Write ymatx to file
```

$\bar{X}$ is formed from X by multiplying the entries of the submatrix H by $\theta\ \Delta t$.

The coupling matrix Q and diffusion matrix H correspond to those in THERM; the porosity matrix S has a similar structure to H and is of minor importance. The programming of subroutine `stiff` is a straightforward extension of that in THERM; in this case we write the the augmented matrices `xmatx,ymatx` to file for each element in turn. X, in `xmatx`, is turned into $\bar{X}$ in the assembly subroutine; this is done to avoid having to repeat `stiff` each time the timestep Δt is changed. The global right-hand-side of the timestepping equation above, is assembled in the subroutine `newrhs`.

In CONSL.FOR we have introduced skyline storage, more efficient than band storage – see Chapter 8 for details. The subroutine `mask` constructs the pointer array needed.

For the equation solution, note that the matrix $\bar{X} + Y$ is symmetric, but is not positive definite because the diagonal entries of H are negative. Thus the symmetric Choleski decomposition cannot be used; instead we use the decomposition $L.D.L^T$ where L is lower triangular with 1's down the diagonal, and D is diagonal. The decomposition and substitution subroutines are `ldldec` and `ldlsub`; see Chapter 8 for full details of the coding.

As mentioned in the previous program THERM, the standard quadrilateral element for this application is an 8-noded serendipity element in which the pore pressure variable is interpolated linearly, using d.o.f.s which exist only at the four corner nodes. A further complication, is that we wish to allow elements representing a structure (in which there are no pore pressure d.o.f.s) to be used in the mesh. Then corner nodes on an interface between a soil element and a structure element will have (u, v, p) degrees of freedom, but the third d.o.f. will be ignored when constructing the stiffness matrix for the structure element. This is coped with by checking the material type of the element, in `stiff`; if `lmtyp(l) = 1` it is a structural element, and we use `ndof = 2`.

A suitable timestepping régime for use in CONSL is as follows. (The various parameters defining it are read in from the final line of the data-file – see section 2.10 above). A special timestep `dtone` is used for the first solution (starting from $u(0) = 0$). With the dual-level element used in CONSL, it is possible to take `dtone = 0.0`. Timestepping begins with the initial timestep `dtint`; five steps are taken using this timestep. The timestep is then multiplied by the factor `ftime`, and used for a further five timesteps. At the end of each set of 5 steps, the user is prompted to say whether s/he wishes to continue timestepping. If `ftime` is taken as 2.0, this regime will give roughly equally-spaced points along the time axis on a logarithmic plot of displacement against time, as is often used. Moreover, the stiffness matrix is constant if the timestep Δt is unchanged, so it only needs to be decomposed in the first of each set of 5 timesteps. (The user will observe that after the first step, the subsequent steps are performed much more quickly.) There is therefore no `solve` subroutine; the routines `ldldec` and `ldlsub` are called directly from the main program. Timestepping stops when the user chooses, or when the input maximum time `ttend` is reached. Results are appended to the `.out` file after each set of five timesteps, and when the program ends. These result sets can be viewed consecutively using FELVUE.

To enable the user to know how consolidation is progressing, the user is invited at run-time to input the node numbers of five nodes at which displacements and pore pressures will be written to screen at each step. The user should choose these nodes using the 'Pick node by mouse' option in the pre-processor, before running CONSL. Suitable nodes are those lying on the surface under loaded areas, or down the centreline of a load. See the example in Chapter 9.

7.4 PLAST.FOR and PLADV.FOR: elasto-plasticity analysis

The theory of Mohr-Coulomb elasto-plasticity is provided in the Appendix. For elasto-plastic material elements, five property values must be prescribed: in addition to the Young's modulus and Poisson ratio, the third component is the triaxial stress factor k and the fourth is the yield strength σ_c (denoted by `triax` and `sigc`). (The relation of these parameters to cohesion c and angle of friction ϕ is given in the Appendix). The PLAST program assumes associated flow, so that the resulting elstoplastic stiffness matrix is symmetric and can be solved by the same routines as in ELAST, namely band storage and Choleski decomposition. The subroutine `mask`, introduced in CONSL.FOR, is however used to determine the halfbandwidth `nhbw` for the mesh. See below for the storage and solution routines used in PLADV.FOR. The 8-noded serendipity quadrialteral element is used, as in ELAST.

Should PLAST can be generalised to non-associated flow, a solver for non-symmetric systems (such as the Gaussian elimination used in THERM.FOR, or the AFRONT subroutine in VPLAS should be used.

The load must be applied incrementally, and within each increment a stress relaxation iteration seeks equilibrium. The load increment loop terminates at statement 200 in the main program. Within this, the iteration for equilibrium starts at statement 100. Subroutine `check` evaluates the residual norm after each iteration, and if this is less than `gtoler` (set in the main program at 10^{-4}) the iteration is taken as having converged. For greater flexibility, the user may wish to input `gtoler` and other tolerances used, as extra parameters in the datafile, when preparing it with the pre-processor (see §2.9, block 2).

A new subroutine `stress` checks for yield, and updates the stresses; a new subroutine `residu` calculates the residual or out-of-balance forces. Yield at a Gauss point ij of element l is denoted by setting mohrg(ij,l) to 1; if mohrg(ij,l)=0, the material is still behaving elastically.

In the program, material type `lmtyp(l)=1` is reserved for purely elastic material, and the plasticity parameters and yield criterion are only used for elements `l` where `lmtyp(l)=2,3,...`

The Gauss point stresses are stored in `strsg` as in ELADV, and separate arrays keep track of the accumulated load applied so far (`accld`), and the total load to be applied (`totld`), in addition to the current incremental load in `aslod`.

The yield criterion used is the Mohr-Coulomb criterion

$$F(\vec{\sigma}) \equiv \sigma_1 - k\sigma_3 - \sigma_c > 0. \tag{7.7}$$

The yield function F is evaluated in subroutine `ycrit`. The flow vector $\vec{a} = \partial F/\partial\sigma$ are found in `flowv`, and the matrix of second derivatives in `flowm`. These are stored in `avect` and `dbds` respectively. If an alternative yield function was to be used, these subroutines would need to be changed.

To avoid complicating the plasticity program code, a separate 'main engine' PLADV has been developed containing alternative methods of storage and equation solution. A parameter `ISOLA` is input at run-time to decide the solver used, as follows:

- ISOLA = 0 for direct solution using Choleski ($L.L^T$) decomposition and skyline storage;
- ISOLA = 1 for direct solution using $L.D.L^T$ decomposition and skyline storage;
- ISOLA = 2 for iterative solution using conjugate gradients with diagonal preconditioning;
- ISOLA = 3 for Incomplete Choleski preconditioning with conjugate gradient solution.

These solvers, and the additional parameters requested for the iterative solvers, are described in more detail in Chapter 8. PLADV uses skyline rather than band storage. There is a trade-off between space efficiency and CPU time required; PLADV with ISOLA=0 runs more slowly than PLAST.

7.5 VPLAS: time-dependent elasto-viscoplasticity analysis

It will be found that the elasto-plasticity program PLAST is not very robust for large-size problems. This is due to the sensitivity of the equilibrium iteration, and the restriction to associated flow. A much more robust program results from using viscoplasticity theory, where the stress-equilibrium iteration is replaced by a time-stepping algorithm. It is also desirable to allow non-associated flow, in which an angle of dilation is included. Such a program is given in VPLAS.FOR, which also includes a sophisticated inner iteration which attempts to reach local stress equilibrium at yielded Gauss points. In line with the recommendation that VPLAS be used for solving practical plasticity problems, a frontal solver has been included, which is very space-efficient. The symmetric solver `front` has already been described in ELADV.FOR above.

See the Appendix for Mohr-Coulomb elasto-viscoplasticity theory. The VPLAS program is developed from PLAST, and the variables described in the previous section apply, with some additions. Thus, the material property sets need, in addition to E and ν, the triaxial stress factor k and yield strength σ_c as in PLAST. The fifth component is the flow rate γ, denoted `gamma`.

The VPLAS program has been generalised to allow for non-associated flow. For this, the sixth component is the dilation factor α (denoted by `dilat` in `stiff`), related to the angle of dilation ψ. If $\psi = \phi$ (so that $k = \alpha$) the flow rule is associated, and the resulting stiffness matrix is symmetric; the frontal solver FRONT is used. If $0 \leq \psi < \phi$, the flow is non-associated, and the stiffness matrix becomes nonsymmetric. In this case, a frontal solver adapted for nonsymmetric matrices is needed. This is provided in the subroutine `afront`. Now the relation between the dimensioning parameters MFRON and MSTIF is MSTIF=MFRON*MFRON, since the full matrix of active variables must be stored. Note that in either `front` or `afront`, some stored data can be used in re-solutions, even if the stiffness matrix has changed, since its structure is still the same. This is exploited by setting IRSOL=1 in re-solutions. Further details of the frontal solvers is given in Chapter 9.

The viscoplastic D matrix is calculated in the subroutine `dmatvp`. The H matrix involved in the theory is calculated in subroutine `hmat`. Subroutine `strain` tests for yield, and calculates the viscoplastic strain velocities (stored in the array `vivel`). Subroutine `steady` decides if these are small enough throughout the mesh to say that equilibrium has been reached.

Chapter 8

Global matrix storage, and equation solvers

When a finite element program runs, it is the solution of the global matrix equation

$$Ku = f \tag{8.1}$$

which takes most of the time, and it is the storage of K which takes most of the space resources. Hence, much attention has been given to algorithms for the compact storage of K, and for efficient solution algorithms. These topics are often given little attention in f.e.m. textbooks, however, because they are seen as peripheral to the main f.e. theory. As FELIPE has been developed to help users understand the practical programming of the f.e. method, a wide range of storage and solution algorithms have been used in the 'main engines', and will be described in this Chapter. Throughout the chapter, we will be considering the solution of (8.1), where K has NTOTV rows and columns.

8.1 Compact storage of K

As we have seen in `POISS` and `ELAST`, the standard f.e. program needs to assemble the global stiffness matrix K from the element matrices written to scratch file by subroutine `stiff`. But storing the whole $n \times n$ matrix K would be impractical. Luckily, we can make use of the fact that K will be **banded**, as you have seen in the graphical demonstration of the assembly process in PREFEL, with zeros in entries outside the band (which lies either side of the diagonal of the matrix). In many applications K will also be **symmetric**, so that we only need to store the lower (or upper) half of the band, together with the diagonal itself.

8.1.1 Symmetric band storage

The simplest example of band storage is in **POISS.FOR**, since in this application there is only one degree of freedom per node; hence, the equation corresponding to the ith node is in the ith row of the stiffness matrix equation. The stiffness matrix will thus have NPOIN rows.

The size of the band is defined by the **bandwidth** NBW. This defined as the smallest number for which: $K_{i,j} = 0$ for all i, j satisfying $|i - j| >$ NBW. If K is symmetric, we will store the lower half of the band, i.e. all entries $K_{i,j}$ for which $i \geq j$. The **halfbandwidth** NHBW is defined by NBW = 2*NHBW + 1.

Since we only get a nonzero entry in $K_{i,j}$ if nodes with global numbering i and j are contained in the same element, the halfbandwidth NHBW can be calculated as the maximum over all elements l, of the difference in node numbers for the nodes in l. In POISS.FOR, this calculation is done at the start of subroutine `assmb`. We will therefore use NPOIN rows of GSTIF (one row for each node in the mesh) and NHBW+1 columns. We denote the latter value by NBAND, and dimension GSTIF at the start of POISS.FOR as
DIMENSION GSTIF(MPOIN,MBAND)
where the parameters MPOIN and MBAND are set in the PARAMETER statement.

Then, on row i we store only $k_{i,i-\text{NHBW}}, \ldots, k_{i,i}$, for $i = 1, \ldots,$NTOTV. We store these entries in an array GSTIF with NTOTV rows (where NTOTV is the total number of degrees of freedom) and (NHBW+1) columns. The mapping from K to GSTIF is:

$$K_{i,j} \rightarrow \text{GSTIF(I,J-I+NHBW+1)} \tag{8.2}$$

for $j = i - \text{NHBW}, \ldots, i$.

Then the diagonal of the matrix (entries where $i = j$) will be stored in the final column of the array. You can see the entries of the element stiffness matrices being placed in GSTIF, in subroutine `assmb` of `POISS.FOR`.

The mapping is slightly more complicated in **ELAST.FOR**, where there are two degrees of freedom per node. Here, the u and v d.o.f.s of node n (i.e. displacements in the x and y directions at node n) are global degrees of freedom numbers $i_n + 1$ and $i_n + 2$, where $i_n = 2(n - 1)$. You can see this mapping being used at the start of subroutine `assmb`. After making this adjustment, the local d.o.f.s are mapped to GSTIF by the formula in the previous section. At the start of `assmb`, the halfbandwidth is found by finding the position of the first non-zero entry in row I: this is stored in the array NMINVC. Thus, if NMINVC(I)=MI, it means that $k_{i,mi} \neq 0$ but $k_{i,j} = 0 \text{for} j = 1, \ldots, mi - 1$. Then the halfbandwidth NHBW is given by $\max_{i=1,\text{NTOTV}}(i - mi)$.

It will be seen that the size of the bandwidth, and hence the amount of storage required, will depend on the order in which the nodes of the mesh have been numbered. For this reason it is very important to use the element renumbering facility in PREFEL

when producing the `.dat` file for input to the 'main engine', to obtain a sensible numbering of the elements — and then to re-number the nodes accordingly. Note that the bandwidth does not depend on the element numbering directly.

8.1.2 Non-symmetric band storage

When the matrix K is nonsymmetric, the whole band must be stored. This is the case in **THERM.FOR**. Here, the array GSTIF has 3N rows (as there are 3 degrees of freedom per node) and NBW columns (NBW is the bandwidth). In this program we introduce the use of arrays NVAR and NDOF, both of size NPOIN. NDOF(N) stores the number of degrees of freedom at node N. NVAR(N) tells us which rows of the global matrix K these degrees of freedom correspond to: if NVAR(N)=IN, then the i'th d.o.f. of node N corresponds to row IN+i. NVAR is formed from NDOF by NVAR(N) = $\sum_{i=1}^{N-1}$NDOF(i). These arrays are used in other 'main engines', especially where there are variable degrees of freedom per node.

The mapping from K to GSTIF is again

$$K_{i,j} \rightarrow \text{GSTIF(I,J-I+NHBW+1)}. \tag{8.3}$$

Now, however, there are NBW columns, and the diagonal entries of K lie down the (NHBW+1)'th column.

8.1.3 Skyline storage

While band storage is considerably better than storing the whole matrix, it still leaves a large number of zero entries within the band. A more efficient storage system is skyline storage, where we decide separately for each row where we need to start the storage.

This storage technique is used in the soil consolidation program **CONSL.FOR**. As with band storage, the amount of storage used depends on the numbering of the nodes in the mesh. We again use the array NMINVC(I) to tell us the column where the first non-zero entry on row I of K lies. Then (as K will by symmetric in CONSL) we store only the entries from this point up to the diagonal entry, in GSTIF. GSTIF now becomes a one-dimensional array, and we use a pointer array LPOINT(I) to tell us where the I'th row of K begins. That is, the diagonal entry $K_{i,i}$ will be stored as GSTIF(LPI), where LPI=LPOINT(I), for I=1 to NTOTV. The dimension of GSTIF is then LPOINT(NTOTV), which we denote by NSTIF. The parameter MSTIF is used to dimension GSTIF at the start of the program. The arrays NMINVC and LPOINT are set up by a subroutine `mask` in CONSL.FOR. LPOINT(I) can be found from LPOINT(I-1) by
LPOINT(I) = LPOINT(I-1) + I - NMINVC(I) + 1
since there are I-NMINVC(I)+1 entries to store on row I. We start with LPOINT(1)=1.

The mapping from the lower diagonal of K to GSTIF is now

$$K_{i,j} \rightarrow \text{GSTIF(LPOINT(I) - I + J)} \tag{8.4}$$

if NMINVC(I)≤J≤I. This mapping is performed by a function `gpos`; if $K_{i,j}$ lies outside the envelope of K which is being stored, `gpos` returns a value of zero. Otherwise,

$$K_{i,j} \rightarrow \text{GSTIF(GPOS(LPOINT,NTOTV,I,J))}. \tag{8.5}$$

8.2 Equation solution algorithms

As with the algorithms for compact storage of K described above, various equation solution algorithms are used in the 'main engines' to solve

$$Gu = f, \tag{8.6}$$

and will be introduced here, carefully graded from the simple to the more complex. We denote the number of rows and columns by n.

8.2.1 Gaussian elimination

The simplest matrix solution algorithm is the well-known **Gaussian elimination**, in which elementary row operations are applied systematically to G – and simultaneously to the right-hand-side vector – to get zeroes in the entries below the diagonal; the resulting upper triangular system can then be solved by back-substitution. This algorithm, which does not require that the matrix is symmetric, is used in the 'main engine' THERM.FOR. The subroutine `solve` calls the subroutines `reduce` and `backsu` in turn, to perform the reduction (to upper triangular form) and back-substitution phases.

This algorithm is represented in the following fragments of pseudo-code. first we show the reduction phase, in which the lower triangular matrix L is formed:

```
C...Reduction:
      do 20 i=1,n-1
        pivot = G(i,i)
        do 10 k=i+1,n
          fact = G(k,i)/pivot
          do 5 j=i+1,n
            G(k,j) = G(k,j) - fact*G(i,j)
    5     continue
          f(k) = f(k) - fact*f(i)
   10   continue
   20 continue
```

Here, G(i,j) indicates the (i,j)'th entry of G; no modification has been made to represent the compact storage of G. If this pseudo-code is compared with the code in subroutine `reduce` of THERM.FOR, it will be seen how the mappings to the band storage of G in GSTIF (as described in the previous section for non-symmetric band storage) are implemented. Also, the start and end values in the do loops are modified, because we only need to perform these loops for entries within the band. These factors in translation from the pseudo-code given in the Manual, to the actual code in the 'main engines', should be borne in mind for when studying the following algorithms.

The solution is completed by the back-substitution phase, performed in subroutine `backsu` of THERM.FOR:

```
c...back-substitution phase
      u(n) = f(n)/G(n,n)
      do 10 i=n-1,1, step -1
        pivot = G(i,i)
        do 5 j=i+1,n
          f(i) = f(i) - G(i,j)*u(j)
    5   continue
        u(i) = f(i)/pivot
   10 continue
```

8.2.2 Symmetric Choleski decomposition

We now consider an algorithm closely related to Gaussian elimination, and used in the Basic-level programs `POISS` and `ELAST`. A **Choleski decomposition** of a positive definite matrix K consists of a lower triangular matrix L (with 1's down the diagonal) and an upper triangular matrix U, such that $K = LU$. If K is symmetric, we modify this to finding a lower triangular matrix L such that $K = LL^T$. (Now we no longer have 1's down the diagonal of K) Then our equation to be solved becomes $L(L^T u) = f$. Once the decomposition has been performed, we solve the equation $Ly = f$ by forward substitution, to find an intermediate vector y; this becomes the right-hand-side in the final stage, which is to solve $L^T u = y$ by back-substitution. This final stage is identical to the back-substitution stage of Gaussian elimination.

The entries in L must be found systematically. The first one which can be found is $l_{1,1}$, since it will be seen by forming the matrix LL^T and comparing it with K, that $k_{1,1} = {l_{1,1}}^2$. Once $l_{1,1}$ is known, we could calculate all the entries below this in the first column. However, we will work in a row-based manner, and find $l_{2,1}$, use this to find $l_{2,2}$, before moving on to the 3rd row to find $l_{3,1}, l_{3,2}, l_{3,3}$ — then move on to the 4th row, and so on. This row-based approach matches the row-based skyline storage algorithm described earlier.

This Choleski decomposition algorithm is used in the Basic-level programs POISS

and ELAST, and can be found in subroutine `chodec`. The pseudo-code for this decomposition, applied to the matrix G, is as follows:

```
c                        T
c...decompose G into L.L
      i = 1
      aii = G(1,1)
      clii = sqrt(aii)
      G(1,1) = clii
c...decompose each row in turn
      do 20 i=2,n
c...work along the row
        do 10 j=1,i-1
          cljj = G(j,j)
          aij = G(i,j)
          sum = 0.0
          do 5 k=1,j-1
            gik = G(i,k)
            gjk = G(j,k)
            sum = sum + gik*gjk
    5     continue
          clij = (aij - sum)/cljj
          G(i,j) = clij
   10   continue
c...finally find the diagonal entry on the row
        aii = G(i,i)
        sum = 0.0
        do 15 k=1,i-1
          clik = G(i,k)
          sum = sum + clik*clik
   15   continue
        clii2 = aii - sum
        clii = sqrt(clii2)
        G(i,i) = clii
   20 continue
```

Note that as each entry in L is found, it is written to G, so that the corresponding entry of the stiffness matrix gets overwritten. At the end of the algorithm, G contains the entries of L in its lower triangle.

In implementing this algorithm in a 'main engine', we must use a mapping to convert G(i,j) to a storage space in GSTIF, as described in the earlier section on storage techniques.

The system $Gu = f$ can now be rewritten $LL^T u = f$, and is solved in two stages:

- solve $Ly = f$ using forward substitution;
- solve $L^T u = y$ using back-substitution.

The pseudo-code for this, on which subroutine `chosub` in POISS.FOR is based, is:

```
c...forward substitution: solve Ly = f
      y(1) = f(1)/G(1,1)
      do 10 i=2,n
        bi = f(i)
        sum = 0.0
        do 5 k=1,i-1
          gik = G(i,k)
          ak = y(k)
          sum = sum + gik*ak
    5   continue
        clii = G(i,i)
        y(i) = (bi - sum)/clii
   10 continue
c                                      T
c...back-substitution phase: solve L x = y
      u(n) = y(n)/G(n,n)
      do 20 i=n-1,1, step -1
        yi = y(i)
        sum = 0.0
        do 15 k=i+1,n
          gki = G(k,i)
          ak = u(k)
          sum = sum + gki*ak
   15   continue
        clii = G(i,i)
        u(i) = (yi - sum)/clii
   20 continue
```

In program CONSL.FOR, we use the same solution algorithm, but modify its Fortran coding to take account of the skyline storage technique, described above. When row-based skyline storage is used with a Choleski decomposition, the algorithm given above for the final back-substitution phase (solve $L^T u = y$) is rather inefficient. This is because it works with rows of L^T, which are columns of L, the entries of which are scattered throughout the row-based storage array GSTIF. It can easily be modified, however, so that once an entry u_i is found, this is eliminated from all equations from $i-1$ up to row 1, which involves working up the column of L^T. The modified algorithm, replacing the last phase of the previous pseudo-code, is:

```
c...back-substitution phase: solve L x = y
c...We can do this in a column-based manner!
      do 20 i=n,1, step -1
        yi = y(i)
        clii = G(i,i)
        xi = yi/clii
        u(i) = xi
c...once we have found x_i, transfer the term involving this, to
c...the r.h.s. in all earlier equations
        if (i.eq.1) goto 20
        do 15 j=i-1,1, step -1
          clij = G(i,j)
          y(j) = y(j) - clij*xi
   15   continue
   20 continue
```

This form of the routine is also used in PLADV, where skyline storage is also employed.

8.2.3 Frontal solution algorithm

The storage of a global stiffness matrix, even using a compact method such as skyline storage, still requires a large amount of space for complex finite element meshes. Researchers have therefore looked for ways of solving $Gu = f$ without having to assemble G fully. One way of doing this is to use iterative solution algorithms, which will be discussed in the next section. However, we first look at a direct solution method which is widely used in commercial f.e. programs. This is called the **frontal method**. A full description is beyond the scope of this manual (see Chapter 10 for recommended texts), but essentially it is a form of Gaussian elimination in which the element assembly process is combined with the solution. First, the mesh topology is examined to identify the element at which each degree of freedom last appears. Then, entries in element stiffness matrices are read in element-by-element, and assigned space in an array of active variables. Once the last appearance of a d.o.f. has occurred, that variable is eliminated, and the space thus freed up is used for a new variable. Hence, there will be a 'front' of active variables passing through the matrix. When this elimination phase is completed, the back-substitution phase can occur.

The coding for this algorithm, for the case of a symmetric matrix, can be seen in subroutine `front` of the advanced elasticity program `ELADV.FOR`. I am indebted to Dr David Naylor for permission to use this subroutine. A number of global one-dimensional arrays have to be used, but the algorithm is still more space-efficient than skyline storage, and so `ELADV` should be used for elasticity analyses with complex meshes. There are comment lines in the subroutine, giving some guidance as to how the algorithm works.

The algorithm can be adapted to the case of a non-symmetric matrix G, in which case approximately twice as much storage will be needed. My own adaptation of `front` for the nonsymmetric case, is the subroutine `afront` in the elasto-viscoplasticity program `VPLAS.FOR`. This program handles non-associated plastic flow, which results in non-symmetric stiffness matrices.

The amount of storage needed depends on the numbering of the elements, not the nodes, as the algorithm is dependent on the order in which the elements are assembled into the front. The subroutine makes a preliminary run-through of the elements to check that sufficient storage is provided; this is defined by the parameter MFRON at the start of the program. The entries of G are allocated spaces in the one-dimensional array GSTIF, which is of dimension MFRON. As the assembly process is handled within `front`, there is no `assmb` subroutine within `ELADV` and `VPLAS`.

8.2.4 Iterative solution methods

The standard iterative method for solving $Gu = f$, G symmetric positive definite, is the **preconditioned conjugate gradients** (PCG) algorithm. This may be summarised as follows:

$$\text{Initialization} \tag{8.7}$$
$$v^{(0)} = 0 \tag{8.8}$$
$$r^{(0)} = f \tag{8.9}$$
$$k = 0 \tag{8.10}$$
$$z^{(0)} = C^{-1}r^{(0)} \tag{8.11}$$
$$p^{(0)} = z^{(0)} \tag{8.12}$$
$$\text{Begin iteration} \tag{8.13}$$
$$\alpha_k = \frac{(z^{(k)}, r^{(k)})}{(Gp^{(k)}, p^{(k)})} \tag{8.14}$$
$$v^{(k+1)} = v^{(k)} + \alpha_k p^{(k)} \tag{8.15}$$
$$r^{(k+1)} = r^{(k)} - \alpha_k G p^{(k)} \tag{8.16}$$
$$z^{(k+1)} = C^{-1}r^{(k+1)} \tag{8.17}$$
$$\beta_k = \frac{(z^{(k+1)}, r^{(k+1)})}{(z^{(k)}, r^{(k)})} \tag{8.18}$$
$$p^{(k+1)} = z^{(k+1)} + \beta_k p^{(k)} \tag{8.19}$$
$$k = k + 1 \tag{8.20}$$
$$\text{Next iteration} \tag{8.21}$$

This form of the PCG algorithm is taken from Bartelt(1989) – see Chapter 10 for

details pf recommended texts. Here, (u, v) denotes the inner product of vectors u and v. The matrix C is the preconditioning matrix. The algorithm is terminated when the norm of the residual vector $r^{(k+1)}$ is less than TOLER times the norm of f; then the solution $u = v^{(k+1)}$.

The simplest form of preconditioning is **diagonal preconditioning**, in which $C = \text{diag}(G)$. This algorithm is implemented in the Basic-level program FRAME.FOR. The solution is performed in subroutine `pcgsol`, the pseudo-code for which is:

```
c...solve Kv=g using diagk as preconditioner
c...multiplications K.v are performed element-by-element,
c...reading the element matrices of K from the scratch file
c
c...initial guess;
      do 2 i=1,n
       v(i) = 0.0
       r(i) = g(i)
       z(i) = 0.0
    2 continue
c
c...z = {Kdiag}^-1 * r
      do 15 i=1,n
        z(i) = r(i)/diagk(i)
   15 continue
c
      do 16 i=1,n
        p(i) = z(i)
   16 continue
      ztr = 0.0
      do 17 i=1,n
        zi = z(i)
        ri = r(i)
        ztr = ztr + zi*ri
   17 continue
      call norm2(mtotv,ntotv,r,rnorm0)
      itscg = 0
      write(6,*)'PGSOL: iteration',itscg,',norm = ',rnorm0
c
   25 itscg = itscg + 1
c
c...create y which is G*p
      call ematgv( p , y )
      pty = 0.0
      do 28 i=1,n
        pty = pty + p(i)*y(i)
```

```
   28 continue
      alpha = ztr/pty
      do 30 i=1,n
       v(i) = v(i) + alpha*p(i)
   30 continue
c
      do 40 i=1,n
        r(i) = r(i) - alpha*y(i)
   40 continue
c
      call norm2(mtotv,ntotv,r,rnorm)
c
      write(6,*)'       iteration',itscg,',norm = ',rnorm
c
      if(rnorm.lt.cgtol) goto 200
c
c...z = {Kdiag}^-1 * r
      do 45 i=1,n
        z(i) = r(i)/diagk(i)
   45 continue
c
      ztr0 = ztr
      ztr = 0.0
      do 60 i=1,ntotv
        ztr = ztr + z(i)*r(i)
   60 continue
      beta = ztr/ztr0
c
      do 130 i=1,ntotv
       p(i) = z(i) + beta*p(i)
  130 continue
c
      goto 25
c
  200 continue
      write(6,*)'Solution converged after ',itscg,' iterations.'
```

With the diagonal preconditioner (stored in the array `diagk`), the formation of $C^{-1}v$ is achieved by simply dividing the elements of v by the corresponding elements of `diagk`. The array `diagk` is assembled from the element stiffness matrices in the subroutine `assdk`. The global matrix-vector multiplication Gv is performed element-by-element, reading the element stiffness matrices from the scratch file, in subroutine `ematgv`; thus, there is no assembly or storage of the global matrix G.

Diagonal preconditioning works surprisingly well, considering its simplicity. If a more

sophisticated preconditioner is required, the most popular currently is to perform an **Incomplete Choleski** (IC) factorization of G. This is essentially a symmetric Choleski decomposition of G into LL^T, as described earlier, but avoiding some or all of the 'fill-in' which occurs within the envelope of G. That is, we may set $L(i,j) = 0$ instead of $L(i,j) = l_{i,j}$ if $G(i,j) = 0$. To compensate this, the calculated $l_{i,j}$ is instead added to the diagonal term $L(i,i)$.

Both diagonal and IC preconditioning are implemented in the advanced plasticity program PLADV.FOR. The form of IC preconditioning there is adaptive: the fill-in is avoided if the magnitude of $l_{i,j}$ is less than a user-specified multiple (`filtol`) of the diagonal term. PLADV.FOR uses the skyline storage technique for G. The subroutine `inchol` in PLADV.FOR performs this IC decomposition. The pseudo-code below should be compared with that for subroutine `chodec`:

```
c...Incomplete Choleski decomposition
      small = 1.e-8
c...count the amount of fill-in which occurs
      kfill = 0
      khole = 0
      i = 1
      a11 = G(1,1)
      cl11 = sqrt(a11)
      G(1,1) = cl11
c...decompose each row in turn
      do 20 i=2,n
        aii = G(i,i)
c...we will not fill in if the term is smaller than filtol
        filtol = factor*abs(aii)
c...work along the row
        do 10 j=1,i-1
          cljj = G(j,j)
          aij = G(i,j)
          sum = 0.0
          do 5 k=1,j-1
            sum = sum + G(i,k)*G(j,k)
    5     continue
          clij = (aij - sum)/cljj
c...here is the 'Incomplete' part
          gij = G(i,j)
          if (gij.eq.0.0) then
            khole = khole + 1
            if (abs(clij).lt.filtol) then
c...avoid fill-in: add i,j term to diagonal instead
              G(i,i) = G(i,i) + clij
              goto 10
```

```
            endif
c...fill-in will occur
            kfill = kfill + 1
          endif
          G(i,j) = clij
   10   continue
c...finally find the diagonal entry on the row
        aii = G(i,i)
        sum = 0.0
        do 15 k=1,i-1
          clik = gstif(i,k)
          if (clik.eq.0.0) goto 15
          sum = sum + clik*clik
   15   continue
        clii2 = aii - sum
        clii = sqrt(clii2)
        G(i,i) = clii
   20 continue
      write(6,*) 'No. of holes found in G matrix = ',khole
      write(6,*) 'No. of holes filled-in = ',kfill
```

As might be expected, there is a trade-off between the amount of fill-in which is prevented, and the effectiveness of the resulting preconditioner. The example datafile `pltun4.dat`, used with the solution options in PLADV, is discussed in Chapter 9.

Note that this is a very simple form of IC algorithm. There have been many improved algorithms proposed in recent years, and users wishing to apply this technique in practice are advised to search specialist journals for the lastest developments.

The reader will have noticed that additional tolerance parameters must be specified when an iterative solver is used. In PLADV, the iterative solvers are chosen by setting the parameter ISOLA to 2 or 3:

- ISOLA = 0 for direct solution using Choleski ($L.L^T$) decomposition and skyline storage;
- ISOLA = 1 for direct solution using $L.D.L^T$ decomposition and skyline storage;
- ISOLA = 2 for iterative solution using conjugate gradients with diagonal preconditioning;
- ISOLA = 3 for Incomplete Choleski preconditioning with conjugate gradient solution.

If ISOLA = 2 or 3 is chosen, the convergence tolerance `cgtol` must be input; a value of 10^{-6} is suitable. (An interesting approach is to make the tolerance adaptive: use

a greater tolerance in the early stages of the plasticity equilibrium iteration, to avoide unnecessary PCG iterations for a solution which is itself only an approximation, and a smaller tolerance for the final iterations). Then, if ISOLA=3, the fill-in tolerance `factor` should be input. If `factor` = 0.0 is chosen, then fill-in will always occur, and so Choleski decomposition will be 'complete'; in this case, the preconditioner of G will be G itself, and the PCG iteration will converge immediately. See the example of `pltun4.dat` in the next Chapter, for experience in choosing `factor`.

8.3 Handling of fixed and specified degrees of freedom

The correct way to handle fixed or specified degrees of freedom in a finite element solution, is to delete those rows from the global system of equations, and in the remaining rows to transfer the terms containing these known values onto the right-hand-sides, before solving for the remaining unknown d.o.f.s. This is however very cumbersome to program, in general. the exception is with the frontal solution method, in which the degrees of freedom are dealt with in turn; it will be found that specified d.o.f.s are treated in this way in `front` and `afront` in ELADV.FOR and VPLAS.FOR.

The alternative method for imposing nodal fixities, used in ELAST.FOR, is sometimes called a 'big spring' approach. To impose the condition that $u_k \approx 0.0$ for some k, we add a large number (10^{20}) to the corresponding diagonal term in GSTIF; this is done in subroutine `fixdof`, called by `solve` before calling `chodec`. This will force the value of u_k arising from the solution process to be very small, and we later set such small values to zero.

Fixed values of u_k which are non-zero (e.g. specified displacements, or the specified values of the potential for nodes lying on Dirichlet boundaries) are dealt with similarly. If we want to ensure that $u_k = g$, we multiply $K_{k,k}$ by 10^{12}, and set the corresponding entry in the r.h.s. vector as $F_k = g * K_{k,k}$. This process can be seen in `fixdof` in POISS.FOR.

When a displacement degree of freedom is fixed, there will be a reaction force in this direction at the node, to maintain equilibrium. This can be found after the main solution, by calculating the r.h.s. for the corresponding row. This is done in subroutine `reactn` in ELAST.FOR and FRAME.FOR; these reactions are written into the load vector ASLOD, which is written in the result file.

Chapter 9

Example datafiles

The FELIPE package (demonstration version and full version) contains a set of example input datafiles (with .dat extension) to illustrate the features of the different 'main engines' described in Chapters 4–7. In general, they have been created in a series: test1.dat contains a coarse mesh created in PREFEL , and this is then refined, and loads etc. added, in test2.dat, test3.dat, etc. The resulting example files which are input into the appropriate 'main engine', will now be described. The names of the files start with two letters indicating the 'main engine' which should be used:

1. POISS.FOR – Files podemo.dat, pohex2.dat, pohex3.dat
2. ELAST.FOR – Files eldemo.dat, elcyl4.dat, elbrac2.dat
3. FRAME.FOR – Files framexa1.dat, framexb1.dat
4. ELADV.FOR – Files eladbm1.dat, eladcv2.dat, eladft3.dat
5. PLAST.FOR – File plcyl4.dat
6. PLADV.FOR – File plcyl4.dat
7. VPLAS.FOR – File vpcyl4.dat
8. THERM.FOR – Files thermln2.dat, thermqd2.dat
9. CONSL.FOR – File conslex3.dat

9.1 Sample datafiles for POISS.FOR: solving Poisson's equation over an H-shaped region

File pohex1.dat contains Part A mesh details for a mesh of three linear quadrilaterals. The coefficients a_x and a_y are set at 1.0 in the material properties section.

In file pohex2.dat the nodes on the left-hand vertical edge have been given Dirichlet boundary fixities; the mesh has then been refined, and doubled by reflection to the right and upwards, and finally triangulated, to create an H-shaped region of width 4.0 and heaight 3.0, containing 384 linear triangles, with Dirichlet boundaries on the outer vertical sides. Part B data has been added, fixing the variable U at 0.0 on the left-hand Dirichlet boundary, and at 60.0 on the right-hand boundary. The other edges act as insulated boundaries.

The program POISS.FOR solves the quasi-harmonic equation, using the coefficients a_x, a_y specified in the data file and with the right-hand-side function equal to 1.0. The result of running POISS.FOR with pohex2.dat as input, is provided in the output file pohex2.out, which can be viewed using FELVUE.

The file pohex3.dat is produced from pohex2.dat by adding a radiating boundary on the lower edge of the crossbar of the H shape, with parameters QBAR=0.0 and ALPHA=4.0. THe effect of this can be seen in the contour plot of U, and the flowlines, in pohex3.out.

An example mesh involving fewer than 50 nodes and 50 elements, which can be run in the demonstration version of POISS.EXE, is provided in podemo.dat. This models only the lower half of the H-shaped region in pohex2.dat, with a coarser mesh. Because of the top boundary being an insulated boundary, it is the same problem which is being solved, as with the full H-shaped mesh.

9.2 Sample datafiles for ELAST.FOR:

9.2.1 Thick cylinder, compressed on outer rim

File elcyl1.dat defines a coarse mesh comprising just one 8-noded element, representing one quarter of a thick cylinder: inner radius 1.0 and outer radius 10.0. The coordinates of the midside nodes of the curved edges are (0.707,0.707) and (7.071,7.071); this is able to model a true quarter-circle to within $1E = 200,000 KN/m^2$ and $\nu = 0.3$. The nodes on the vertical and horizontal edges are constrained in the x and y directions respectively, to provide by symmetry a solution for a complete circular annulus.

In elcyl4.dat the mesh has been refined to 100 elements, and a compressive loading applied to the outer edges of the outermost row of elements. The top element (adjoining the vertical boundary) has an applied pressure of $35 KN/m^2$ (traction set 2); the neighbouring element has a 'ramp' loading which drops from $35 KN/m^2$ to $30 KN/m^2$ (traction set 1), and then the next elements in have a uniform pressure of $30 KN/m^2$. The ramp loading up to $35 KN/m^2$ is reproduced at the horizontal boundary. ELAST.EXE should be run with this datafile, using the plane strain option at run-time. The resulting displacements and stresses can be viewed in FELVUE in file elcyl4.out.

The data-file eldemo.dat is a coarse mesh (just 12 elements) for a thick cylinder subject to a uniform load on the outer $10KN/m^2$ on the outer rim. This can be run with the demonstration version of ELAST.EXE, again in plane strain. If the resulting stress contours are plotted in FELVUE there will be seen a small non-uniformity around the cylinder due to the coarseness of the discretization. Remember that FELIPE does not apply any stress-smoothing, and contours are plotted element-by-element, so that inaccuracies caused by an insufficiently fine discretization are not disguised.

9.2.2 Plane stress example: bracket under tension

File elbrac1.dat shows an initial mesh for analysing the problem of a plastic bracket, 15cm long and 6cm wide, with a hole of diameter 3cm drilled through it, centred 3cm from the right-hand end. The bracket is 1cm thick, and its analysis is a case of plane stress. The left-hand end is fixed to the wall. A metal pipe is inserted through the hole (so that the bracket holds it to the wall), and we wish to examine the stresses and displacements caused in the bracket if the pipe attempts to push away from the wall. Our mesh only models half the problem: there is reflection about the centreline $y = 0$. The refined mesh (using the Automatic refine option, then further refining around the pipe) is contained in elbrac2.dat, which shoulf be input into ELAST.EXE, run using the plane stress option.

We model only the left half of the pipe cross-section, since in practice the right half will not exert any influence on the bracket; it will become detached from the plastic, which we cannot cope with in our program. We apply a uniform load of $10^5 KN/m^2$ (equivalent to 10KN per square centimetre, as we are working in cm), pushing leftwards on the centreline of the pipe. The material properties used are:
For the pipe (material set 2): $E = 10^5, \nu = 0.4$
For the bracket (material set 1): $E = 2,500, \nu = 0.2$
The material thickness is set at 1.0 in the 3rd component of each set. A tensile strength of 10 is also specified for the bracket, in the 4th component. The material type (LMTYP) facility is also used: the bracket and pipe elements are assigned material types 1 and 2 respectively.

The resulting deformations and stresses of the bracket are best seen in FELVUE if you choose to plot only those elements with material type 1. The stress concentration is seen to be around the rear of the pipe section (use the Zoom facility to examine these in more detail), and if the "Yield Zones" plotting option (choosing yield code 1) is used, you will see the Gauss points at which the tensile strength has been exceeded. This is largely due to the plastic flowing in behind the pipe, which is not realistic: a better mesh would include a 'wing' of pipe extending part-way around the rear of the half-pipe section.

Note that this is a hypothetical example, and the material property values do not bear resemblance to real materials.

9.3 Sample datafiles for FRAME.FOR

For FRAME.FOR, two simple examples of plane frames are given. File framexa1.dat describes a two-beam frame which is pin-jointed at either end, with a uniform load distributed along the horizontal beam (Note the negative sign for this, according to the way in which the compression-positive sign convention applies to 1D elements – see Chapter 5). The dimensions are actually in inches, and the load is 1000 lbs/ft. The material properties of the beams are:
E = 3×10^7, I = 100, A = 10
This example is actually worked example 4.14 in the Schaum series text (Buchanan 1994), and the solution for nodal displacements and rotations can be checked against this. The conjugate gradient solver converges in 5 iterations, using diagonal preconditioning. In viewing the results in FELVUE, use the "Plot 1D elts" option and answer 'y' to the question whether the 1D elements represent beams. Note that the "View displts" option only displays 2D elements, so a blank screen will result from this option. The "Plot d.o.f.s" option will also not apply for frame analyses.

The second example, in framexb1.dat, is also based on a problem in Buchanan(1994), and Imperial units (inches, p.s.i.) are used. It consists of a three-beam frame, pin-jointed at its right-hand end and free to roll in the horizontal plane at the other, loaded with a uniform pressure on its middle member, and with a point-load acting downwards at the mid-point of the left-hand beam. The PCG iteration converges (to a tolerance of 10^{-5}) after 19 iterations. The results can be viewed in FELVUE in the same way as with the previous example.

9.4 Sample datafiles for ELADV.FOR

The datafiles provided for ELAST can also be used with the Advanced-level program ELADV. The following additional datafiles demonstrate some of the extra capabilities of ELADV:

9.4.1 Beam part-embedded in elastic block

Following on from the beam examples above, datafile eladbm1.dat demonstrates that 1D beam elements can be used in the same mesh as 2D plane elements in ELADV. The mesh shows a vertical beam with its base embedded in an elastic block which is anchored at its base. It is loaded with a distributed force applied normally on the top element. The beam elements used here are the three-noded quartic elements, which are more closely compatible with the serendipity quadrilaterals than the standard 2-noded cubic elements.

In FELVUE, the deformed beam, together with the deformed boundary of the block, can be seen using the "View 1D elts" option with an exaggeration factor of 1.0. The "Displacements" option displays the deformation of the block itself, without the 1D elements. A mesh with more refinement would produce better results for stress contours, although the plot of Gauss point stress crosses is reasonable; principal stresses drawn in blue are tensile.

9.4.2 Footing on infinite soil mass

The file eladft3.dat shows a very coarse mesh for a concrete footing on a semi-infinite soil mass. It is included to demonstrate how different element types can be combined in the same mesh. As well as serendipity quadrilaterals, there are 6-noded triangles and infinite elements (including a doubly-infinite element at the mesh corner). The load is applied as a normal surface traction on the top surface of the concrete footing.

Again, a finer discretization would produce more meaningful stress contours.

9.4.3 Cavern excavated in pre-stressed rock mass

The example illustrating some of the extra advanced elasticity facilities in ELADV.FOR is of a cavern excavated in a pre-stressed rock mass of infinite extent. Half of the cavern is modelled in file eladcv1.dat, consisting of a dome roof (modelled by a curved Serendipity element as in the thick-cylinder example) above a 2m deep-by-1m wide rectangular space. Beyond a layer of serendipity elements surrounding the cavern boundary lie mapped infinite elements (including a 3-noded corner (doubly-infinite) element) representing the infinite rock mass. Elastic parameters are $E = 5,000 KN/m^2$ and $\nu = 0.2$. Because infinite elements are used all around the mesh, there is no need to fix any nodes in the y-direction; only nodes on the centreline are constrained in the x direction, for symmetry. This is a plane strain analysis.

In eladcv2.dat the mesh has been refined, and the excavation boundary around the cavern edge defined. A uniform stress field of $\sigma_x = 20, \sigma_y = 40$ is stipulated for all the elements, which have been assigned material type 2 as required in ELADV. A further feature is that a small 6-noded triangular element has been added after refinement to "cut off the corner" at the bottom of the cavern. The "add elements" feature in "modify mesh" was used, and by leaving blank the coordinates of the new node (on the midside of the hypotenuse), these were linearly interpolated by the pre-processor. The node was then moved using the "change coordinates" option, to produce a rounded profile; this greatly improves the accuracy of the stress field around the corner singularity.

The deformations and stress field resulting from the unloading around the cavern boundary, can be seen by viewing eladcv2.out. A realistic swelling of the cavern roof and floor, where areas of greatest deviator stress occur, is noted. (The initial exaggeration

factor is too large, and a factor of 20 should be tried, in conjuction with the Zoom facility, to see the deformation around the cavern wall). If running ELADV with this example, notice that the equation solution is performed by the frontal algorithm FRONT.

9.5 PLAST.FOR, VPLAS.FOR, PLADV.FOR: thick-cylinder example revisited

The file plcyl4.dat, for use with PLAST.FOR, is derived from elcyl1.dat, but written at Advanced level with additional material properties: triaxial stress factor $k = 3.0$ and yield strength $\sigma_c = 20.0$ defining an elasto-plastic Mohr-Coulomb rock (stored as material property components 3,4). The associated-flow plasticity algorithm often requires very small load increments in order to achieve convergence, when the applied loads are asymmetrical, so for this example a uniform compressive load of $25KN/m^2$ over the outer circumference is applied. This load is applied in 5 increments. Viewing the output in plcyl4.out, you can see the drop in major stress on the innermost two rows of Gauss points, which have yielded plastically; the "yield zones" plot (choosing yield code 1) will show this zone. Remember that an associated flow rule is used, which keeps the elasto-plastic stiffness matrix symmetric.

The file vpcyl4.dat is an extension of elcyl4.dat for the viscoplasticity program VPLAS.EXE. Here, in addition to the material parameters k and σ_c given above for PLAST, the flow parameter $\gamma = 0.01$, and dilation l are defined in material property components 5,6; in this case $l = 1.0$, i.e. a non-associated zero-dilation flow rule. VPLAS.FOR uses the frontal algorithm subroutine AFRONT, adapted for nonsymmetric matrices, in this solution. The loading is similar to that in elcyl4.dat, with $25KN/m^2$ around the outer boundary, rising to $35KN/m^2$ at each end; the viscoplasticity algorithm involving timestepping is much more stable than the plasticity algorithm involving iteration, especially if a non-associated flow rule is used, and it converges happily even with the non-uniform loading. In addition to an incremental loading regime with 5 load increments, a timestepping regime is defined in the last line of the data set. The Crank-Nicholson time discretisation parameter $\theta = 0.5$ has been used. Compare the stresses, displacements and yield zones output in vpcyl4.out, with those from the elasticity and elasto-plasticity analyses.

The advanced plasticity program PLADV.FOR can also be run with the datafile plcyl4.dat. Results will be identical to those from PLAST.FOR, but can be obtained using a selection of different solvers. If the preconditioned conjugate gradient solver (ISOLA=2) with diagonal preconditioning is used, each solution takes about 180 PCG iterations (for a convergence tolerance of 10^{-6}). If the same convergence tolerance is used with the Incomplete Choleski preconditioner (ISOLA=3), the performance depends on the amount of fill-in which occurs. this is controlled by the fill-in factor which is also specified at run-time. if a factor of 0.1 is chosen, none of the 47,052 'holes' within the envelope of K are filled in, and convergence takes 56 iterations. If a factor of 10^{-4} is

chosen, then about 2,500 'holes' get filled in, and the iteration speeds up, needing 24 iterations to convergence. With a fill-in factor of 10^{-6}, about 50% of the 'holes' (some 25,000) are filled-in, and only 5 iterations are needed to converge.

9.6 THERM.FOR: cooling fin

To illustrate the plane thermoelasticity application, we consider the problem of a rectangular cooling fin, 0.5m wide and 1m tall, anchored at its base to a hot vessel, temperature 100 degrees, while its other three sides are exposed to the air, temperature 20 degrees. The elastic constants are $E = 10^6, \nu = 0.4$, and this is a plane stress analysis with a thickness of 0.5cm. The coefficients of thermal conductivity and of thermal expansion, entered as material properties 4 and 5, are 1.0 and 5.0 respectively (these were chosen arbitrarily).

The fin is modelled with meshes of 4-noded linear and 8-noded quadratic quadrilateral elements, in datafiles thermln2.dat and thermqd2.dat respectively. Examining the displacements in FELVUE, you will see how the fin has expanded under this thermal loading (choose an exaggeration factor of 1,000 to get a proper view), and can also view the distribution of tensile stresses generated in the metal. Proceeding to the d.o.f. plots, a plot of d.o.f. 3 will display the temperature field in the fin.

As mentioned in Chapter 7, the results using quadratic elements are good, but in the results from the mesh of linear elements the well-known 'hourglass' instability is very evident. To avoid this instability, a two-level discretization should be used, with quadratic discretization of displacement d.o.f.s but only a bilinear interpolation for the temperature using values at corner nodes. This type of element is implemented for the soil consolidation application CONSL.FOR.

9.7 CONSL.FOR: footing on saturated soil

In this problem, a concrete block (one element, material type 1) rests on a rectangular mesh of elements (material type 2) representing a soil mass. Elastic parameters are $E = 10,000 KN/m^2, \nu = 0.4$ for the concrete block, and $E = 100 KN/m^2, \nu = 0.2$ for the soil. In addition, the soil material set has component 3 set to 0.001, representing the porosity of the interstitial pore fluid. The horizontal and vertical soil permeability coefficients are set at 0.004.

In the refined mesh in file conslex3.dat the "modify mesh" option has been used to write at Advanced level and to add an extra degree of freedom at corner nodes only, for elements with material type 2. Note that corner nodes on the interface between block and soil elements, have the third d.o.f., but the CONSL program ignores this when processing the stiffness matrix for the block element (which has material type 1). This

extra d.o.f. – the pore pressure – has been fixed at zero on the free surface $y = 0$, apart from at nodes underneath the impermeable concrete block; the fixity code has been used for this. A uniform normal surface traction of $10KN/m^2$ is applied to the top of the concrete block.

The timestepping régime involves making an initial solution with timestep `dtone=0.0`, the results of which are written to file. Then fives timesteps will be made with a timestep of `dtint=1.0`, after which the timestep will be doubled (`ftime=2.0`) and a further five steps made, and so on. Timestepping will halt when `ttend=100.0` is reached, unless the user has chosen to halt earlier. A fully implicit timestepping algorithm is used (`theta=1.0`).

In order to judge how consolidation is progressing, the user can enter at run-time the numbers of five nodes at which displacement/pressure results will be written to screen at each timestep. The nodes to be used should be chosen by viewing the mesh of conslex3.dat in PREFEL and using the "Pick node by mouse" facility (You will need to Zoom first!). In the present mesh, suggested nodes are:
Node 151 (on the centreline, on top surface of block)
Node 145 (on the centreline, on interface between block and soil)
Node 185 (on surface at one corner of the block)
Node 139 (on centreline, soil depth of 0.1m)
Node 137 (on centreline, soil depth of 0.3m)

If CONSL.EXE is run with datafile conslex3.dat, it will be seen that after 15 timesteps, when $t = 35.0$, consolidation is virtually complete, and the timestepping may be ended. The program appends the results after each set of 5 timesteps, to the file conslex3.out. Running FELVUE with this file, first the undrained deformation is displayed (an exaggeration factor of 10 for the displacements is appropriate. Note that the soil elements close to the footing bulge unrealistically – a finer refinement in this part of the mesh is needed). You can see the undrained pore pressure distribution in the soil, by proceeding to the d.o.f. plots, and plotting d.o.f. 3 for material type 2 (soil elements). Remember to answer 'y' to the question, if this d.o.f. exists only at corner nodes. Again, it is necessary to zoom to see the excess pore pressures under the footing, in detail. (A consequence of the bulging is that a zone of negative pore pressure arises in these elements).

Proceeding, past the option of printing PostScript files, you are now invited to proceed to the next result set. This will be the solution at $t = 5.0$. View the displacements, stresses and pore pressures as before. Repeat the process for the results at $t = 15.0$ and $t = 35.0$.

Chapter 10

Further Reading, and suggested exercises

There is a sharp divide between books describing the finite element method from a mathematical point of view, and those concentrating on its practical application for the solution of engineering problems; this short bibliography deals with the latter category.

Having said that, the 'bible' for all f.e.m. practitioners, whether mathematicians or engineers, is: is *The Finite Element Method in Engineering Science*, OC Zienkiewicz, McGraw-Hill. Those new to FEM may find the early all-in-one-volume editions (e.g. 2nd ed, 1971) more useful for an overview than the latest one (by Zienkiewicz & Taylor).

Good introductory textbooks for engineers which are clearly-written and relatively cheap include:

- *Finite Element Analysis (Schaum outline series)*, GR Buchanan (McGraw-Hill 1994, ISBN 0-07-008714-8) contains worked examples, and covers beam elements and 2D thermoelasticity as well as elasticity. It does not contain any programming as such, but is very useful as a reference book for the formulae needed in programming. There are, however, a lot of typographical errors.
- *A Practical Introduction to Finite element Analysis*, YK Cheung & MF Yeo (Pitman 1979, ISBN 0-273-01080-8). Includes useful 'tricks' such as the 'big spring' method for fixed degrees of freedom.
- *Finite Element Analysis*, MJ Fagan (Longman 1992, ISBN 0-582-02247-9). No programs, but some glossy photos and a chapter on f.e. packages.
- *Engineering Stress Analysis*, DN Fenner (Elliss Horwood 1987).
- *Finite Element Programming*, E Hinton & DRJ Owen (Academic Press 1977). Particularly relevant to geotechnical applications, and includes a description of the

frontal solution algorithm. The programming style (naming conventions, etc.) is similar to that in FELIPE. The book covers beam theory, but the element described is not the standard beam element but an isoparametric 3-noded element using only the quadratic 1D shape functions.

- *Introduction to the Finite Element Method*, N Ottosen & H Petersson (Prentice Hall 1992, ISBN 0-13-473877-2). No programs, but covers programming aspects such as node renumbering.

- *Finite Elements for Structural Analysis*, W Weaver Jr & PR Johnston (Prentice Hall 1984, ISBN 0-13-317099-3). Fortran listings not included, but available from the authors.

The above texts include Fortran program listings unless otherwise indicated, and cover the standard linear engineering situations (plane stress/strain, axisymmetric elastics, beams, plates and shells).

For elasto-plasticity, especially in geotechnical situations, I would recommend *Finite Elements in Plasticity*, DRJ Owen & E Hinton (Pineridge Press 1980, ISBN 0-906674-05-2). As with the authors' introductory text, the programming style is very similar to that used in PLAST.FOR and VPLAST.FOR. There is a serious need for a more up-to-date version of this book. The only modern practical textbook covering plasticity, soil-structure interaction and other advanced geotechnical problems, is *Programming the Finite Element Method (3rd ed)*, IM Smith & DV Griffiths (Wiley 1998, ISBN 0-471-96542-1). Earlier editions have programs in Fortran77, but the 3rd edition uses Fortran90.

A compendium of algorithms for nonlinear analyses is *Nonlinear Finite Element Analysis of Solids and Structures, Vol 1*, MA Crisfield (Wiley 1991, ISBN 0-471-92956-5). This and other titles by Crisfield cover iterative methods such as conjugate gradients, but the 'bible' for PCG in finite lement solutions, and for Incomplete Choleski factorization in particular, is *Finite Element Solution of Boundary Value Problems*, O. Axelsson & V.A. Barker (Academic Press 1984, ISBN 0-12-068780-1). The PCG algorithm quoted in Chapter 8 is from *Finite element Procedures on Vector/Tightly Coupled Parallel Computers*, P. Bartelt (Verlag der Fachvereine Zürich, 1989, ISBN 3-7281-1742-0).

Finally, if you want to explore the underlying mathematical theory, two user-friendly presentations in paperback are:

- *Finite Elements and Approximation*, OC Zienkiewicz & K Morgan (Wiley 1983, ISBN 0-471-89089-8).

- *Finite Element Analysis and Applications*, R Wait & AR Mitchell (Wiley 1985, ISBN 0-471-90678-6).

The mapped infinite elements available in FELIPE are described in the later editions

of Zienkiewicz's *The Finite Element Method in Engineering Science*, and a detailed exposition is *Infinite Elements*, P Bettess (Penshaw Press 1992, ISBN 0-9518806-0-8).

10.0.1 And finally...

Since the Fortran source code for the 'main engines' such as `POISS.FOR` and `ELAST.FOR` are provided, students may explore the finite element method by making modifications to these programs, re-compiling and seeing the effects when solving standard problems. For example, the effect of the integration rule used can be investigated by simple modifications in the subroutines calculating the element stiffness matrices. Thus, students could be asked to change the subroutine `stiff` in `POISS.FOR` (Section 3.3.1) to use a three-point integration rule — the midside rule:

$$\int_0^1 \int_o^{1-\eta} F(\xi,\eta)\ |\mathbf{J}|\ d\xi\ d\eta \doteq \frac{1}{6}(F(\frac{1}{2},0) + F(\frac{1}{2},\frac{1}{2}) + F(0,\frac{1}{2})). \tag{10.1}$$

In `ELAST.FOR`, students could change the subroutine `gauss` to use the 3×3 Gauss integration rule, derived from the 3-point rule:

$$\int_{-1}^{1} f(\xi)d\xi \doteq \sum_{i=1}^{3} w_i f_i \tag{10.2}$$

where $w_1 = w_3 = \frac{5}{9}, w_2 = \frac{8}{9}$, and f_i are the values of f at the Gauss-points $\xi = -\sqrt{\frac{3}{5}}, 0, \sqrt{\frac{3}{5}}$. Test the program on problems with known solution — are the results more accurate or less accurate?

This Manual could be expanded into a textbook providing student exercises and projects. If you would like to see this, please recommend to suitable publishers that they commission such a book!

The FELIPE package is also very suitable as a basis for constructing your own, more complex programs — for example, modifying `VPLAS.FOR` to use more realistic yield criterion, and to handle some of the range of element types and loading conditions used in `ELADV.FOR`. If you need to solve practical problems beyond the range of the FELIPE 'main engines', but you do not have the time or the programming expertise to develop your own program, I am available to act as a consultant, for a negotiated fee; please contact me!

Training courses using FELIPE can also be arranged, either at Brunel University or on site.

I am very interested to hear your views and experiences with using FELIPE, and your suggestions for improvements. Please email me at: `martin@felipe.co.uk` .

Chapter 11

Appendix

11.1 Element types available

11.2 Elasto-plasticity theory

11.3 Elasto-viscoplasticity theory

11.4 Thermoelasticity theory

11.5 Soil consolidation theory