FEAP - - A Finite Element Analysis Program

Version 8.6 Programmer Manual

Robert L. Taylor & Sanjay Govindjee
Department of Civil and Environmental Engineering
University of California at Berkeley
Berkeley, California 94720-1710

Revised January 2020

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

In this part of the *FEAP* manual some of the options to extend the capabilities of the program are described. We begin by describing the utilities provided in *FEAP* for use in data input. Options to add user commands for mesh and command language extensions is then described and finally the method to add an element to the program is described.

1.1 Setting Program Options

The size of problems which may be solved by *FEAP* depends on the amount of memory available in the computer, as well as, solution options used. Memory for the main arrays used to solve problems is dynamically allocated during the solution. Arrays are allocated and deallocated using a system subprogram `PALLOC` or, for user developed modules using subprogram `UALLOC`. Further information on use of these routines is given in Section 3.

The `IPR` parameter in the `feap86.f` module controls the specification of the ratio of `REAL` to `INTEGER` variables. For typical UNIX and PC systems all real variables should be twice as large as integers and `IPR` is set to 2. For systems in which `INTEGER*8` variables are used (set by compiler option) the `IPR` parameter is set to 1. Any error in setting this parameter may lead to incorrect behavior of the program, consequently, do not reset the parameter unless a careful assessment of compiler behavior has been made.

Normally *FEAP* reads each input data line as text data and checks each character for the presence of parameters, expressions, and constants. For very large data sets this parsing of each instruction can consume several seconds of compute time. If all data is normally provided as numerical data, without use of any parameters or expressions, the input time may be reduced by setting the value of the logical variable `COFLG` in `feap86.f` to *false*. *FEAP* will automatically switch to parsing mode if any record contains non-numerical data item. It is also possible to use the `PARSe` and `NOPArse` commands to set the appropriate mode of data input.

In Windows versions it is sometimes desirable to obtain the input file name from a pop-up menu. This is accomplished by setting the parameter `CIFLG` to true.

During the input of plot commands *FEAP* has the option to either set input options automatically (`DEFAult` mode) or to read the values or range of contours to plot. The default mode of operation may be assigned in the `feap86.f` module by setting the variables `DEFAult` and `PROMPT`. Setting `DEFAult` to *true* indicates that all default options are to be set automatically. If `DEFAult` is set *false*, a prompt for contour intervals may be requested by setting `PROMPT` to *true*.

FEAP has options to produce encapsulated PostScript output files in either gray scale of in color. The default mode may be established by setting the variable `PSCOLR` and `PSREVS`. Setting `PSCOLR` true indicates the PostScript files will be in color (unless set otherwise by the `PLOT COLOr` data command. The `PSREVS` variable reverses the color sequence.

The last parameter which may be set in the `feap86.f` module is the level for displaying available commands when the `HELP` command is used while in mesh, solution, or plot mode. *FEAP* contains a large number of commands which are not commonly used by many users. To control the default number of commands displayed to users the commands have been separated into four levels: (0) Basic; (1) Intermediate; (2) Advanced; and (3) Expert. The level to be displayed when using the `HELP` command is given may be set in the integer variable `HLPLEV`. That is, setting:

```
hlplev = 1      ! Intermediate
```

results in commands up to the *intermediate* level being displayed. It is possible to raise or lower the level during execution using the command `MANUal,,level` where level is the numerical value desired.

When developing program modules it is often desirable to have output of specific quantities available (e.g. tracking the change in some parameters during successive iterations. *FEAP* provides for a switch to make the outputs active or inactive during an execution. The switch is named `debug` and placed in

```
integer         ndebug
logical                debug
common /debugs/ ndebug,debug
```

The value of the debug is set true by the solution command `DEBUg` and false by the command `DEBUg,OFF`. Thus, placing code fragments into modules as

```
       if(debug) then
         write(iow,*) 'LABEL',list ...  ! writes to output file
 !  and/or
         write(  *,*) 'LABEL',list ...  ! writes to screen
       endif ! debug
```

This device supplements use of available debuggers on the computer.

1.2 Fortran variable declaration

FEAP has been developed over many years and contains programming style in *Fortran 77*; *Fortran 90* and later versions. Thus, most of the files use the `*.f` extender and not `*.f90`.[1] For the unix version some routines are also written in `C`. User modules may be added using either the syntax for `*.f`, `*.f90` or `*.c`.

The main real and integer variables in *FEAP* are set using the Fortran declarations

```
       real    (kind=8)  :: or it can be real*8
       complex (kind=8)  ::
       integer           ::
       integer (kind=8)  :: ! For use with 'mr' or 'hr'
       character (len= ) ::
       logical           ::
```

In particular we do not recommend the use of

```
       integer (kind=4)  ::
```

which is equivalent to `integer` since this does not permit compiler options to convert to large 64-bit integers. In some instances there are some declaration of 32-bit real variables using

```
       real (kind=4) :: or real*4
```

however, these are mostly for the timing routine, not main variables.

[1]The parallel module also uses files with `*.F` extender for preprocessing by PETSc.

1.3 Uses of Common and Include Statements

FEAP contains many `COMMON` statements that are used to pass parameters and small array values between subprograms. For example, access to the debugging parameter `debug` is facilitated through `common /debugs/`. Users may either place the common statement (as well as data typing statements) directly in the routine or may use an include statement. For debugging the statement would be

```
include  'debugs.h'
```

which during compilation would direct the precompiler to load the current common statement from this file. In *FEAP* all include files have the same name as the common with an added extender `.h`. For example, the common file name `comblk.h` is defined as

```
real (kind=8) :: hr
integer                  mr
common /comblk/  hr(1024),mr(1024)
```

The arrays `hr(1024)` and `mr(1024)` serve to pass all dynamically allocated arrays between subprograms using a pointer array contained in the common array named `np(*)` [or for user defined arrays in `up(*)`] located in the include file `pointer.h`. [2] See Section 3 for more details on use of pointers. All include files are located in the directories `include`.

It is highly recommended that users use include files rather than giving equivalent common statements directly. If later releases of the *FEAP* program revise contents in a common block, it will only be necessary to recompile the user routine rather than change all the common statement definitions.

[2]The values `1024` are necessary to ensure loops on arrays using pointers directly are considered as long.

Chapter 2

DATA INPUT AND OUTPUT

FEAP includes utilities to perform input and to output small arrays of data. Users are strongly encouraged to use the input utilities but often may wish to use their own utilities to output data.

2.1 Parameters and Expressions

The subroutines `PINPUT` and `TINPUT` are input subprograms used by FEAP to input each data record. They permit the data to be in a free form format with up to 16 items (or 256 characters) on each record, as well as to employ expressions, parameters, and numerical representations for each data item. These routines also should be used to input data in any new program module developed. The PINPUT routine returns data to the calling subprogram in a double precision array. The following statements may be included as part of the routine performing the input.

```
      subroutine xxx(.....)

      include 'iofile.h' ! ior,iow,ilg unit numbers

      logical       :: errck, pinput
      real (kind=8) :: td(5)

   1  if(ior.lt.0) write(*,3000)
      errck = pinput(td, 5)
      if(errck) go to 1
```

The parameters defined in the include file (common block) are:

```
ior   - input  file unit number (if negative, input
        from keyboard)
iow   - output file unit number
ilg   - solution log file unit number
```

If an error occurs during input from the keyboard *FEAP* returns a value of true for the function and a user may reinput the record if the implied loop shown above is used. For inputs from a file, the program will stop and an error message indicating the type of error occurring and the location in an input file is written to the output file.

The input routines return data in a `real*8` array `td(*)`. If any td(i) is to be used as an `integer` or `real*4` quantity, it must be cast to the correct type. That is, the following operations should be used to properly cast the variable type:

```
real (kind=4) :: t
real (kind=8) :: td(5)
integer       :: j
logical       :: errck, pinput

errck = pinput (td, 5)

j = nint(  td(1))  ! Integer assignment
t = float(td(2))   ! Real*4  assignment
```

PINPUT may be used to input up to 16 individual expressions on one input record (each input record is, however, limited to 256 characters).

The routine TINPUT differs from PINPUT by permitting text data to also be input. It is useful for writing user commands or to input data described by character arrays. The routine is used as

```
logical            :: errck, tinput
integer            :: nt, nn
character (len=15) :: text(16)
real      (kind=8) :: td(16)

errck = tinput(text,nt,td,nn)
```

The parameter `nt` specifies the number of *text* values to input and the `nn` specifies the number of *real data* values to input. The value for parameter `nt` or `nn` may be zero. Thus the use of

```
errck = tinput(text,0,td,nn)
```

is equivalent to

```
        errck = pinput(td,nn)
```

Text variables may be converted to numerical (`REAL*8`) form using the subroutine call

```
        call setval(text,nc,td)
```

where `text` is a string with `nc` characters and `td` a `REAL*8` variable. The text string can contain any parameters, expressions or numerical constants which evaluate to a *single* value.

2.2 Array Outputs

Two subprograms exist to output arrays of integer and real (double precision) data. The routine `MPRINT` is used to output real data and is accessed by the statement:

```
        call mprint( array, nrow, ncol, ndim, label)
```

where `array` is the name of the array to print, `nrow` and ncol are the number of rows and columns to output, `ndim` is the first dimension on the array, and `label` is a character label which is added to the output. For example the statements:

```
        real (kind=8) :: aa(8,6)
    . . .
        call mprint( aa(2,4), 2, 3, 8, 'AA')
```

outputs a 2×3 submatrix from the array `aa` starting with the entry `aa(2,4)`. The output entries will be ordered as the terms:

```
        aa(2,4)  aa(2,5)  aa(2,6)
        aa(3,4)  aa(3,5)  aa(3,6)
```

The `MPRINT` routine adds row and column labels as well as the character label.

The routine `NZPRINT` is used to output the upper non-zero block of a real array and is accessed by the statement:

```
        call nzprint( array, nrow, ncol, ndim, label)
```

where all parameters are identical to those for `MPRINT`.

The routine `IPRINT` is used to output integer data and is accessed by the statement:

```
        call iprint( array, nrow, ncol, ndim, label)
```

where all parameters are identical to those for `MPRINT` except the array must be of type integer.

The routine `CPRINT` is used to output complex (kind=8) data and is accessed by the statement:

```
        call cprint( array, nrow, ncol, ndim, label)
```

where all parameters are identical to those for `MPRINT` except the array must be of type complex.

The routine `LPRINT` is used to output logical data and is accessed by the statement:

```
        call lprint( array, nrow, ncol, ndim, label)
```

where all parameters are identical to those for `MPRINT` except the array must be of type logical.

Chapter 3

ALLOCATING ARRAYS

Dynamic data allocation is accomplished in *FEAP* by defining addresses in pointers contained in the common block defined in `pointer.h`. This common block contains pointers `np` for standard program arrays and `up` for user defined arrays and has the form

```
integer          num_nps       , num_ups
parameter       (num_nps = 400 , num_ups = 200)

integer (kind=8) :: np           , up
common /pointer/    np(num_nps)   , up(num_ups)
```

Each pointer is an offset relative to the address of a `REAL*8` array `hr(1)` or an `INTEGER` array `mr(1)` defined in a blank common

```
real (kind=8) :: hr
integer                  mr
common /comblk/  hr(1024),mr(1024)
```

which is placed in the file `comblk.h` in the `include` directory. The pointers 64-bit length (*i.e.*, `integer (kind=8)`) allows access to all of the computer memory. *The arrays 'hr' and 'mr' are used to establish addresses only and not to physically store data.* This mechanism permits references to elements in arrays which have positions relative to `hr` or `mr` that may be after or before 1. Thus, *FEAP* **must** be compiled **without** strict array bound checking. Size of problems is limited only by the available memory in the computer used.

When using 64-bit pointers users must be careful to always define the address of an array in a calling statement to also be 64-bits in length. For example use of

```
integer    :: ioff
 ...
ioff = np(111) + numnp
call submat( hr(ioff), ...)
```

would cause an error since the pointer `ioff` is only 32 bits in length. To avoid this problem it is necessary to either declare `ioff` to be 64-bits long as

```
integer (kind=8) :: ioff
```

or use one of the *FEAP* include files `p_int.h` (defining the integer type array `fp(10)`) or `p_point.h` (defining the integer type scalar `point`).

Using this scheme permits direct reference to either `real*8` or `integer` arrays in program modules without need to pass arrays through arguments of subprograms. A subprogram `PALLOC` controls the allocation of all standard arrays in *FEAP* defined by the `np` pointers and a subprogram `UALLOC` permits users to add allocation for their own arrays defined by the pointers `up`. The basic use of the routines is provided by an instruction

```
setvar = palloc(number,'NAME',length,precision)
```

or

```
setvar = ualloc(number,'NAME',length,precision)
```

where `setvar`, `palloc` and `ualloc` are logical types, `number` is an integer number of the array, `NAME` is a 5 character name of the array, `length` is the number of words of storage needed for the array, and `precision` is the type of array to allocate (1 for `integer` and 2 for `real*8` types). Upon initial assignment of any array its values are set to zero. Thus, if the array is to be used only once it need not be set to zero before accumulating additional values. If the array is to be reused or resized (see below) it must be reinitialized prior to accumulating any additional values. Use of these subprograms controls the assignment of memory space for all arrays such that no conflicts occur between `hr` and `mr` referenced arrays. Each routine which makes direct reference to an allocated array using a pointer (e.g., `hr(np(43))` or `mr(up(1))`) must contain include files as

```
include  'pointer.h'
include  'comblk.h'
```

As an example for the use of the above allocation scheme consider a case where it is desired to allocate a real (double precision array) with length `NUMNP` (number of nodes in mesh) and an integer array with length `NUMEL` (number of elements in mesh). The parameters `NUMNP` and `NUMEL` are contained in `COMMON /CDATA/` and available using the include file `cdata.h`. The new arrays are defined using the temporary names `TEMP1` and `TEMP2` which have numerical locations '111' and '112', respectively.[1] The two arrays are allocated using the statements

```
        setvar = palloc( 111, 'TEMP1', numnp, 2 )
        setvar = palloc( 112, 'TEMP2', numel, 1 )
```

where the last entry indicates whether the array is `REAL*8` (2) or `INTEGER` (1). These arrays are now available in any subprogram by specifying the `pointer.h` and `comblk.h` include files and referencing the arrays using their pointers, e.g., in a subroutine call as:

```
        include  'pointer.h'
        include  'comblk.h'
          ...
        call subname ( hr(np(111)) , mr(np(112)) .... )
```

Note the use of `hr(*)` and `mr(*)` for the double precision and integer references, respectively. Also, the use of the pointers avoids a need to include the array reference until it is needed in a computation.

A short list of the mesh arrays available in *FEAP* is given in Table 3.1, for solution arrays in Table 3.2, and for element arrays in Table 3.3. The names of all active arrays in any analysis may be obtained using the `SHOW,DICTionary` solution command.

The array `IX(nen1,numel)` is used to store basic information for each element in the mesh related to the nodal connections and material data requirements. In addition, arrays `IE` and `IEDOF` define additional information required to process each element. Tables 3.4, 3.6 and 3.7 describe the use of individual entries in the arrays `IX`, `IE`, and `IEDOF`, respectively.

The subprograms `PALLOC` and `UALLOC` may also be used to destroy a previously defined array. This is achieved when the length of the array is specified as zero (0). For example, to destroy the arrays defined as `TEMP1` and `TEMP2` the statements

[1] See the subprogram `palloc.f` in the `program` directory for the names and numbers of existing arrays.

NAME	Num.	dim 1	dim 2	dim 3	Description
ANG	45	`numnp`	-	-	Angle
D	25	`ndd`	`nummat`	-	Material parameters
F	27	`ndf`	`numnp`	2	Force and Displacement
ID	31	`ndf`	`numnp`	2	Equation nos. (1) and B.C. (2)
IE	32	`nie`	`nummat`	-	Element control, dofs, etc.
IX	33	`nen1`	`numel`	-	Element connections
T	38	`numnp`	-	-	Temperature
U	40	`ndf`	`numnp`	3	Solution array
VEL	42	`ndf`	`numnp`	`nt`	Solution rate array
X	43	`ndm`	`numnp`	-	Coordinates

Table 3.1: Mesh Array Names, Numbers and Sizes

NAME	Num.	dim 1	dim 2	dim 3	Description
CMASn	n+8	`compro`	-	-	Consistent Mass
DAMPn	n+16	`compro`	-	-	Damping
JPn	n+20	`neq`	-	-	Profile pointer
LMASn	n+12	`neq`	-	-	Lump Mass
TANGn	n	`maxpro`	-	-	Symmetric tangent
UTANn	n+4	`maxpro`	-	-	Unsymmetric tangent

Table 3.2: Solution Array Names, Numbers and Sized

NAME	Num.	dim 1	dim 2	dim 3	Description
ANGL	46	`nen`	-	-	Angle
LD	34	`nst`	-	-	Assembly nos.
P	35	`nst`	-	-	Element vector
P	35	or	`ndf`	`nen`	Element vector
S	36	`nst`	`nst`	-	Element matrix
TL	39	`nen`	-	-	Temperature
UL	41	`ndf`	nen	6	Solution array
XL	44	`ndm`	nen	-	Coordinates

Table 3.3: Element Array Names, Numbers and Sizes

NAME	Description
`IX( 1 ,e)`	Global node 1
...	to
`IX(nen ,e)`	Global node nen
`IX(nen+1,e)`	H1 history data pointer
`IX(nen+2,e)`	H2 history data pointer
`IX(nen+3,e)`	H3 history data pointer
`IX(nen+4,e)`	Lagrange multiplier tag
`IX(nen+5,e)`	Lagrange multiplier data pointer
`IX(nen+6,e)`	Time integrator: 0=implicit; > 0=explicit
`IX(nen+7,e)`	Element type: FE ≤ 0; IGA > 0
`IX(nen1 ,e)`	Element material type number
`IX(nen1-1,e)`	Element region number (default = 0); Active region >0; Inactive region <0
`IX(nen1-2,e)`	Active/deactive start
`IX(nen1-*,e)`	Used for element data pointers

Table 3.4: Element connection array `IX` use for element `e`

Number	Shape
0	Undefined
1	Line
2	Triangle
3	Quadrilateral
4	Tetrahedron
5	Hexagon
6	Wedge
7	Pyramid
8	Point

Table 3.5: Element types in `IX(nen+7,e)`

NAME	Description
`IE(1,ma)`	Plot shape dimension (0,1,2,3); 0 = no plot, 1 = line; 2 = surface; 3 = solid.
`IE(2,ma)`	Rigid material number.
`IE(nie ,ma)`	Number history variables/element (NH1 and NH2).
`IE(nie-1,ma)`	Element material type number (`ELMT01 = 1`, etc.).
`IE(nie-2,ma)`	Element material type identifier (default = ma).
`IE(nie-3,ma)`	Offset to NH1/2 history variables (default = 0).
`IE(nie-4,ma)`	Offset to NH3 history variables (default = 0).
`IE(nie-5,ma)`	Number history variables/element (NH3).
`IE(nie-6,ma)`	Finite rotation update number (for PROTxx or UROTxx).
`IE(nie-7,ma)`	Get tangent from element if 0; if > 0 numerically differentiate residual to obtain tangent.
`IE(nie-8,ma)`	Equation number for element Lagrange multiplier.
`IE(nie-9,ma)`	Partition number for element Lagrange multiplier.
`IE(nie-10,ma)`	Global equation number.

Table 3.6: Element control array `IE` use for material number `ma`

NAME	Description
`IEDOF(1,i,ma)`	Degree of freedom 1 for node `i` of material `ma`.
...	to
`IEDOF(ndf,i,ma)`	Degree of freedom `ndf` for node `i` of material.

Table 3.7: Element degree of freedom assignment array `IEDOF` use for material number `ma`

```
        setvar = palloc( 111, 'TEMP1', 0, 2 )
        setvar = palloc( 112, 'TEMP2', 0, 1 )
```

are given. Use of these statements results in the pointers `np(111)` and `np(112)` being set to zero and the space used by the arrays being released for use by other allocations at a later point in the program.

A call to `PALLOC` or `UALLOC` for any previously defined array but with a different non-zero length causes the size of the array to be either increased or decreased.

For user defined arrays specified in `UALLOC` care should be exercised in selecting the alphanumeric `NAME` parameter, which is limited to 5 characters, so that conflicts are not created with existing names (use of the `SHOW,DICT` command is one way to investigate names of arrays used in an analysis) or check the names already contained in the subprogram `PALLOC`.

The subroutine `PGETD` also may be used to retrieve internal data arrays by `NAME` for use in user developed modules. For example, if a development requires the nodal coordinate data the call

```
        integer    :: xpoint, xlen, xpre
        logical    :: flag
        ....
        call pgetd ('X ',xpoint,xlen,xpre,flag)
```

will return the first word address in memory for the coordinates as `xpoint`, the length of the array as `xlen`, and the precision of the array as `xpre`. If the retrieval is successful `flag` is returned as true, whereas if the array is not found it is false. The precision will be either one (1) or two (2) for `INTEGER` or double precision (`REAL*8`) quantities, respectively. Thus, the above coordinate call will return `xpre` as 2 and `xlen` will be the product of the space dimension of the mesh and the total number of nodes in the mesh. The first coordinate, x_1, may be given as

```
        x1 = hr(xpoint)
```

any other coordinates at nodes may also be recovered by a correct positioning in later words of `hr`. For example y_1 is located at `hr(xpoint+1)`. The use of `pgetd` can lead to errors for situations in which the length of arrays changes during execution, since in these cases the value of the pointer `xpoint` can change. For such cases a call to `pgetd` must be made prior to each reference involving `xpoint`. On the other hand, reference using the pointers defined in arrays `NP` or `UP` are adjusted each time an array changes size. However, users must ensure that a calling sequence is not sensitive to a change in pointer. One way pointer changes can still lead to errors is through a program

```
        call subname ( hr(np(111)), mr(np(112)), ....)
```

and then change the length of the array number ‘111’ or ‘112’ in the subroutine.

Chapter 4

USER FUNCTIONS

Users may add their own procedures to facilitate additional mesh input features, to perform transformations or manipulations on mesh data, to add new solution commands, or to add new plot capabilities.

4.1 Mesh Input Functions - `UMESHn`.

To add a mesh input command a subprogram with the name `UMESHn`, where `n` has a value between 0 and 9 must be written, compiled, and linked with the program. The basic structure of the routine `UMESH1` is:

The parameter `TX` is a character array which is assigned by the input and `UPRT` is a logical parameter which is set to false when the `NOPRint` mesh command is given and to true when the `PRINt` command is used (default is true). The common block `UMAC1` transfers the character variable `UCT` to assign the name of the command. The default name is `MESn` where `n` is the same as the routine name number. Assignment of a unique character name (which must not conflict with names already assigned for *mesh input commands*) should be used to replace the `xxxx` shown.

When *FEAP* begins execution it scans all of the `UMESHn` routines and replaces the command names `mes1`, etc., by the user furnished names. Thus, when the command `HELP` is issued while in interactive MESH mode, the user name will appear in the list instead of the default name (note, *FEAP* does not always display all available commands. To see all commands issue the command `MANUal,3` and then the `HELP` command).

```
       subroutine umesh1(tx, uprt)

!-----[--.----+----.----+----.-------------------------------------]
!      Purpose: User defined routine to input mesh data to FEAP

!      Inputs:
!        tx(*)  - Command line input parameter name
!        uprt   - Flag, Output results if true

!      Outputs:
!        none   - Users responsible for outputs to arrays, etc.
!-----[--.----+----.----+----.-------------------------------------]
       implicit  none

       include  'umac1.h'  ! Contains UCT variable
       character (len=15) :: tx(*)*15
       logical            ::  uprt

 !     Set name 'mes1' to user defined
       if(pcomp(uct,'mes1',4)) then
         uct  = 'xxxx'    ! Set user defined command name
        elseif(ucount) then              ! Count elements and nodes

        elseif(urest.eq.1) then          ! Read  restart data

        elseif(urest.eq.2) then          ! Write restart data

       else

 !       User execution function statements follow

       end if

       end subroutine umesh1
```

Figure 4.1: Sample `UMESHn` module

The ability to get array names as shown in Chapter 3 can be used to develop user routines for input of coordinates, element connections, etc. With this facility it is possible to develop an ability to directly input data prepared by other programs which may be in a format which is not compatible with the requirements of standard *FEAP* mesh commands.

4.1.1 Command line `TX` data

It is possible to include up to 8 data items on the command line for user functions. All the data is passed to the `UMESHn` functions by the character array `TX(*)*15` and may be used to control actions in the function. If the information is of type character it may be used directly, however, if it is numeric it must be converted within the `UMESHn` function. before any additional input statements are processed. For example if a user input function has the command line:

```
GETData VALUes 35
```

is developed in the user function `UMESH1` the first argument `GETData` must match the name assigned to `UCT` and will also be in `TX(1)`. The second parameter will be in `TX(2)` and the third in `TX(3)`. To recover the numerical value for the third parameter the statement statements

```
real (kind=8) :: ctl
...
call setval(tx(3),15, ctl)
```

may be used to assign the real value $35.0d0$ to `ctl`. If necessary, the real value for `ctl` can be cast into an integer using

```
itl = nint(ctl)
```

If more than 8 items are desired on the input line it is possible to recover their values from the character string `yyy*256` which has been parsed into columns with width 15 characters. Note that the total number of added words must be 15 items or less (this is imposed by the total of 16 items on any *FEAP* input record). To recover their values the statements

```
include   'chdata.h'
character (len=15) :: lct(15)*15
real      (kind=8) :: rtl(15)
integer            :: itl
```

are added to the user function and the items recovered in the `else` option of the function using the statements:

```
lct(1) = yyy(16:30)
call setval(yyy(31:45),15, rtl(1))
```

would assign `lct(1)` values from the second set of 15 characters and `rtl(1)` to the third set of 15 characters. In this case `lct(1) = tx(2)` and rtl(1) would have the same value as `ctl` above.

If users wish to add more than 10 material models it is possible to use the user function `UMESH` which has the form

4.1.2 Nodal coordinate inputs

A `UMESH` command is useful to input the nodal coordinates and element connectivity from external mesh generation programs. The name of the data set to be read is described by the part of a umesh

```
if(pcomp(uct,'mes*',4)) then
  uct = '.....' ! name should not conflict with any other
```

Often multiple nodal and element data sets are required to completely specify the problem mesh. In some cases each of the data sets have node and element numbers beginning with unity. Alternatively, the data may be given without any node or element number and implicitly begin with unity. In these cases it is necessary to establish a unique number for every node or element. In *FEAP* the `*auto` [see User Manual[1] for details] may be used to create the unique numbers. For single data sets the command is not needed. The descriptions below for nodes and elements describes how to program for this feature.

For the input of nodal coordinate data, the number of nodal items can be determined from the data either by counting the number of items, from a separate record, or from the command data as described above using the `tx` data array. In `FEAP` this is

```
      logical function umesh(cc,tx,prt)

!-----[--.----+----.----+----.-----------------------------------------]
!      Purpose: User mesh command interface

!      Inputs:
!         cc     - User command option
!         tx(*)  - Command line input data
!         prt    - Output if true

!      Outputs:
!         none   - Data stored by user development
!-----[--.----+----.----+----.-----------------------------------------]
      implicit  none

      logical            :: prt,pcomp
      character (len= 4) :: cc
      character (len=15) :: tx(*)

!     Match on 'USER': Add as many checks as desired with 'user'

      if(pcomp(cc,'xxxx',4)) then  ! Provide name for 'xxxx'

        umesh = .true.  ! Activate command
        .....

      elseif(.........

      endif

      end logical function umesh
```

Figure 4.2: Sample UMESH module

established by either reading the mesh data once before the input phase or specifying the actual numbers on the control record. For user mesh modules `umeshn` [`n=0:9`], once known for each data set the number should be returned as:

```
elseif(ucount) then
  unumnp =  "number of nodes in data set"
```

where `unumnp` is found in

```
include   'umac1.h'
```

This allows `FEAP` to determine the total number of nodes in a mesh, even if multiple data sets are used to describe the coordinates.

The actual input of the data may be performed by adding the two include files

```
include   'pointer.h'  ! np(*) pointers
include   'comblk.h'   ! mr(*) and hr(*) arrays
```

and then adding a call as:

```
else
  call unode_xxxx(hr(np(43)),mr(np(190)) ...)
```

where `umesh_xxxx` is a user defined module in which `hr(np(43))` is the location of the nodal coordinate data and `mr(np(190))` is the location of the nodal activation data. By default all the `numnp` coordinates are marked as not defined in this array. The module `unode_xxxx` may be given as:

```
        subroutine unode_xxxx(x, ndtyp, ....)

        implicit   none

        include   'cdata.h'   ! numnp
        include   'sdata.h'   ! ndm
        include   'dstars.h'  ! starnd, starel

        integer       :: ndtype(numnp)
        real (kind=8) :: x(ndm,numnp)
          ....
!       Loop over data set,input local node "n" and set node number
        nod        =  n + starnd    ! For *AUTO data inputs
        x(:,nod)   = ....           ! Input values
        ndtyp(nod) = 0              ! Activate node
        ....                        ! After all data inputs
        starnd = starnd + ..        ! Add number input items
```

The critical part is setting the correct mesh node number and activating the node. By default `starnd` (and `starel`) are zero at the begging of mesh inputs. The `starnd` parameter keeps track of how many total nodes have been described.

4.1.3 Element connectivity inputs

For `UMESH` modules used to input connectivity and material set numbers into the `IX(NEN1,NUMEL)` the type of element should be inserted into the position `IX(NEN+7,*)` for each element. The value to be inserted is shown in the *Feap Value* column of Table 4.1. The table also shows the element forms that will be displayed when using a ParaView output command.

TYPE Parameter	Nodes/ Element	Feap Value	ParaView Value
LINE	2	-1	3
	3	-1	21
TRIAngle	3	-2	5
	6/7	-2	22
	10	-2	-
QUADrilateral	4	-3	9
	8/9	-3	23
	12/16	-3	-
TETRahedron	4	-4	10
	10	-4	24
HEXAhedron	8	-5	12
	20/27	-5	25
	64	-5	-
WEDGe	6	-6	13
PYRAmid	5	-7	14

Table 4.1: Element TYPE specification on connectivity input.

A `umeshn` module may be used to input the nodal connection date in a similar manner to that used for nodal coordinate input. Accordingly, the number of elements in the data set is returned as:

```
      elseif(ucount) then
        unumel =  "number of elements in data set"
```

where `unumel` is found in

```
        include   'umac1.h'
```

This allows FEAP to determine the total number of elements in a mesh, even if multiple data sets are used.

The actual input of the element data may be performed by adding the two include files

```
        include   'pointer.h'  ! np(*) pointers
        include   'comblk.h'   ! mr(*) and hr(*) arrays
```

and then adding a call as:

```
        else
          call uelmt_xxxx(mr(np(33)),) ...)
```

where uelmt_xxxx is a user defined module in which mr(np(33)) is the location of the element connection data. By default all elements are marked as not input by a large negative material set number for each element "e" in ix(nen1,e). The module uelmt_xxxx may be given as:

```
        subroutine uelmt_xxxx(ix, ....)

        implicit   none

        include   'cdata.h'   ! numel
        include   'sdata.h'   ! nen1
        include   'dstars.h'  ! starnd, starel

        integer       :: ix(nen1,numel)
          ....
!       Loop over data set,input local element "e" and set global number
        eg           =  e + starel          ! For *AUTO data inputs
        ixl(1:nel)   = ....                 ! Sets local node  number
        ix(1:nel,eg) = ixl(1:nel) + starnd  ! Sets global node number
        ix(nen1,eg)  = ....                 ! Set material set number
        ix(nen+7,eg) = ....                 ! Set element shape type
        ....                                ! After all data inputs
        starel = starel + ..                ! Add number input items
```

The critical part is setting the correct mesh node and element numbers. using the current starnd and starel values. By default starnd (and starel) are zero at the begging of mesh inputs. The starel parameter above keeps track of how many total elements have been described.

```
      subroutine umani1

!     User defined routine to manipulate mesh data for FEAP

      implicit  none

      include  'umac1.h'  ! Contains UCT variable

!     Set name 'man1' to user defined
      if(pcomp(uct,'man1',4)) then
        uct  = 'xxxx'    ! Set user defined command name

!     User execution function statements follow
      else

      end if

      end subroutine umani1
```

Figure 4.3: Sample `UMANLn` module

4.2 Mesh Manipulation Functions - `UMANIn`.

The `UMANIn` modules, where `n` ranges from 0 to 9, may be used to perform transformations or manipulations on previously prescribed data. These commands appear between the mesh input `END` command and the first `INTE`ractive or `BATC`h solution command. To add a mesh manipulation command a subprogram with the name `UMANIn`, where `n` has a value between 0 and 9 must be written, compiled, and linked with the program. The basic structure of the routine `UMANI1` is:

The common block `UMAC1` transfers the character variable `UCT` for the name of the command. The default names are `MANn` where `n` is the same as the routine name number. Assignment of a unique character name (which must not conflict with names already assigned for *mesh input commands*) should be used to replace the `xxxx` shown.

After *FEAP* completes the input of mesh data it scans all of the `UMANIn` routines and replaces the command names `man1`, etc., by the user furnished names.

The ability to get array names as shown in Chapter 3 can be used to develop user routines for manipulation of the mesh data. For example, if a user has added the specification of information by coordinates it may later be necessary to associate the data with specific node numbers. This can be accomplished using a manipulation command which searches for the node number whose coordinates are closest to the specified location.

```
      subroutine umacr0(lct,ctl)

!     User solution command function

      implicit  none

      include  'umac1.h'     ! Contains the variable UCT

      character (len=15) :: lct*15
      real      (kind=8) :: ctl(3)

!     Set command word
      if(pcomp(uct,'mac0',4)) then
        uct = 'xxxx'

!     User command statements are placed here
      else

      endif

      end subroutine umacr0
```

Figure 4.4: Sample `UMACRn` module

4.3 Solution Command Functions - `UMACRn`.

In a similar manner, users may add solution commands to the program by adding a routine with the name `UMACRn` where n ranges from 0 to 9.

The parameters `LCT` and `CTL` are used to pass the second word of a solution command and the three parameter values read, respectively. Again the name `xxxx` should be selected to not conflict with existing solution command names and will appear whenever HELP is issued.

4.4 Plot Command Functions - `UPLOTn`.

In a similar manner, users may add new plot commands to the program by adding a routine with the name `UPLOTn` where n ranges from 0 to 9.

The parameters `CTL(3)` are used to pass the three parameter values read, respectively. Again the name `xxxx` should be selected to not conflict with existing plot command names and will appear whenever HELP is issued.

```
      subroutine uplot0(ctl)

!     User plot command function

      implicit  none

      include  'umac1.h'     ! Contains the variable UCT

      real (kind=8) :: ctl(3)

!     Set command word
      if(pcomp(uct,'plt0',4)) then
        uct = 'xxxx'

!     User plot command statements are placed here
      else

      endif

      end subroutine uplot0
```

Figure 4.5: Sample `UPLOTn` module

4.4.1 Plot of lines and filled panels

Two plot utilities are available for placing lines on the screen. These are named `DPLOT` and `PLOTL`. The calling form for `DPLOT` is given as

```
      call dplot(s1,s2,ipen)
```

where `s1, s2` are screen coordinates ranging from 0 to 1. Similarly, the calling sequence for `PLOTL` is

```
      call plotl(x1,x2,x3,ipen)
```

where `x1, x2, x3` are coordinates values of the mesh. The value of `ipen` ranges from 1 to 3: 1 starts a filled panel; 2 draws a line from the current previous point to the new point; 3 moves to the new point without drawing a line. If a filled panel is started it must be closed by inserting the statement

```
      call clpan()
```

4.4.2 Plot numbers

Positive and negative integer numbers may be plotted using the module

```
call plabl(n)
```

where `n` is the integer to place in the plot region. Prior to using a move to the plot location should be made using the `dplot` or `plotl` routines described above.

4.4.3 Plot text

Text may be placed in the plot region using the call

```
call pltext(x,y,il,string)
```

where `x, y` are *screen* coordinates and `len` is the length of the `string` of text to place.

4.4.4 Plot colors

Lines are drawn or panels filled in the current color. A color is set using the statement

```
call pppcol(color, switch)
```

where `color` is an integer defining the color number and `switch` should be zero. The color values are given in Table 4.2.

4.5 User Material Models

Users may add material models to elements by appending subprograms `UMATIn` and `UMATLn` (where `n` have values from 0 to 9) to the *FEAP* system. The subprogram `UMATIn` defines the input of parameters used by the model and the subprogram `UMATLn` is called by the element for *each* computation point (*i.e.*, the quadrature point), receives the value of a deformation measure as input and must return the value of stress and tangent moduli as output.

Number	Color	Number	Color
0	Black	10	Green-Yellow
1	White	11	Wheat
2	Red	12	Royal Blue
3	Green	13	Purple
4	Blue	14	Aquamarine
5	Yellow	15	Violet-Red
6	Cyan	16	Dark Slate Blue
7	Magenta	17	Gray
8	Orange	18	Light Gray
9	Coral		

Table 4.2: Color Table for Plots

To activate a user material model the input data for the mesh `MATErial` command must include a statement with `UCON` as the first field. For example in a solid element the command sequence can be

```
MATErial ma
  SOLId
    UCONstitutive xxxx v1 v2 ...
       ! Blank termnation record
```

The role of the `xxxx` and `vi` data will be described in Section 4.5.1.

It is possible to use standard input parameters defined in Tables 5.5 to 5.5, as well, by preceding the `UCON` command with a normal input sequence. For example, if isotropic elastic properties are needed they may be included in the input sequence as

```
MATErial ma
  SOLId
    ELAStic ISOTropic  e  nu
    UCONstitutive xxxx v1 v2 ...
       ! Blank termnation record
```

No standard commands should follow the `UCON` command.

Alternatively, users may input elastic properties as part of their `UMATIn` module. For example, the sample module shown in Figure 4.6 would input the data as

```
MATErial ma
```

```
SOLId
  UCONst E_1d  e  ! e = Young's modulus
     ! Blank termnation record
```

If the user routine does input additional data records (after the `UCON` record) and these are terminated by a blank record, a second blank record will be needed to discontinue material data input for this set. In all cases at least one blank record is always needed to terminate the input of standard options for the material set. Extra blank records may always be used without causing problems

4.5.1 The `UMATIn` Module

A sample module for a user constitutive model is shown in Fig. 4.6. As shown in this figure, the `UMATIn` module has 5 arguments. The name of the constitutive equation to be described is passed in the first parameter `type`. The second parameter passes an array (`vv(*)`) which may be used to define up to 5 parameters for the material model. The example shown above for the `UCON` includes the `type` data as `xxxx` and the array `vv(*)` values as `v1 v2 ....` Users may also provide additional input within the `UMATIn` module using the routines `PINPUT` or `TINPUT` described in Sect. 2.1. The values of user parameters must be saved in the array `ud(*)` (the fourth argument of `UMATIn`). In the current version there are 150 words of double precision values available by default. Additional values may be allocated by assigning a larger value on the control record (first record after the `FEAP` title record). Each material model is assigned a user material number to the return parameter `umat`. This number must be a positive integer. Finally, the number of history parameters to be assigned to each computation (quadrature) point must be returned in the parameter `n1`. Currently, the parameter `n3` may be set but is not available to the user material model. Thus, all history variables must be retained in the `n1` list. Use of history variables is described later as part of the `UMATLn` module.

4.5.2 The `UMATLn` Module

In preparing user material models for *FEAP* it is recommended that the develop be made for a general three-dimensional model. In this way the material can work properly in conjunction with both the two-dimensional plane and axisymmetric solids as well as the general three-dimensional elements.

The `UMATLn` module is used to compute the stress and tangent moduli at time t_{n+1}

```
      subroutine umati1(type,vv, d, ud, n1,n3)
!-----[--.----+----.----+----.-----------------------------------]
!       Purpose: User material model interface

!       Inputs:
!          type    - Name of constitutive model (character variable)
!          vv(*)   - Parameters: user parameters from command line
!          d(*)    - Program material parameter data

!       Outputs:
!          n1      - Number history terms: nh1,nh2
!          n3      - Number history terms: nh3
!          ud(*)   - User material parameters
!-----[--.----+----.----+----.-----------------------------------]
      implicit  none

      include  'iofile.h'
      logical            :: pcomp
      character (len=15) :: type*15
      integer            :: n1,n3
      real      (kind=8) :: vv(5),d(*),ud(*)

!     Specify type of user model

      if(pcomp(type,'mat1',4)) then
        type = 'E_1d'                 ! Specify new name for model

!     Input/output user data and save in ud(*) array

      else

!       Set values of 'n1' if required
        n1   = ...

        write(iow,*) ' User Constitutive Inputs: E = ',vv(1)
        ud(1) = vv(1) ! Parameter from input on command name

      endif

      end subroutine umati1
```

Figure 4.6: Sample UMATI1 module

from the supplied deformation measure at time t_{n+1}. For small strain the supplied deformation measure is the linear strain which is ordered as

$$\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_{11} & \epsilon_{22} & \epsilon_{33} & \gamma_{12} & \gamma_{23} & \gamma_{31} \end{bmatrix}$$

To access the strain at time t_n it is necessary to dimension the strain array as `eps(9,*)`; with `eps(9,1)` providing the t_{n+1} strains and `eps(9,2)` the t_n strain. Note the first dimension is 9, however, only the first six entries are used for the small strain model.

In finite strain the deformation gradient and the displacement gradient at times t_{n+1} and t_n are passed to the `UMATLn` module in the array `f(3,3,4)`. The array `f(3,3,1)` stores the value of the deformation gradient at t_{n+1}; `f(3,3,2)` stores the deformation gradient at the time t_n; `f(3,3,3)` stores the displacement gradient at t_{n+1}; and `f(3,3,4)` stores the displacement gradient at t_n. The displacement gradient $\mathbf{G}$ is given by

$$\mathbf{G} = \mathbf{F} - \mathbf{I}$$

where $\mathbf{I}$ is the unit tensor (or identity matrix). *FEAP* computes the displacement gradient and then adds the identity. Thus using the displacement gradient as much as possible is recommended to avoid round-off when it is very small. It is recommended users study some of the models included in the library to see how developments can be made for various deformation measures.

Although the basic finite deformation measure passed has nine components, both the small and the finite strain user models must return only six components of stress and their associated tangent moduli. The stresses and moduli are returned in a Voigt notation in the order

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{22} & \sigma_{33} & \tau_{12} & \tau_{23} & \tau_{31} \end{bmatrix}$$

The two-dimensional solid elements (located in the directory `./elements/solid2d`) include formulations for plane stress, plane strain, axisymmetric without torsion and axisymmetric with torsion. Table 4.3 describes the stress components used in each formulation and the values of the associated variable `stype` (see Table 5.5) to define each type of analysis.

However, coding a user material model for full 3-d analysis is always recommended. All standard *FEAP* solid elements pass the unused 3-direction strain or deformation measures with a zero value.

A sample for the `UMATL1` module with arguments defined for small deformation is shown in Fig. 4.7 and for arguments defined for finite deformation in Fig. 4.8. This subprogram will be called by many of the elements included within *FEAP* if a user model

Analysis Type	`stype`	Stress components used
Plane Stress	1	σ_{11}, σ_{22}, τ_{12}
Plane Strain	2	σ_{11}, σ_{22}, τ_{12}
Torsionless Axisymmetric	3	σ_{11}, σ_{22}, σ_{33}, τ_{12}
Axisymmetric with Torsion	8	All 6 stress components

Table 4.3: Stress components used in 2-D analyses.

has been specified as part of the `MATE` mesh data (see previous subsection). The user model will not be called for truss, frame, plate, and shell elements which use resultant models to describe behavior. Also, any form which requires a one-dimensional model will not use a `UMATLn` module. The module is designed to compute three-dimensional constitutive models in which the stress and strain are stored as 6-component vectors and the tangent moduli as a 6×6 matrix.

Small deformation models

For small deformation models the strains are passed to `UMATLn` in the argument array `eps(6)` and stored in the order

$$\boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_{11} & \epsilon_{22} & \epsilon_{33} & \gamma_{12} & \gamma_{23} & \gamma_{31} \end{bmatrix}^T$$

where $\gamma_{ij} = 2\,\epsilon_{ij}$ is the engineering shearing strain. Stress and moduli are to be associated with the same ordering and returned in the argument arrays dimensioned as `sig(6)` and `dd(6,6)`, respectively. All real values are in double precision (i.e., `REAL*8`).

When `UMATLn` is called the model `n` will be that which is defined in the module `UMATIn`. Current values of the strains are, as mentioned above, passed in the array `eps(6)` and the trace of the strain in the parameter `theta`. Thus,

$$\theta = \epsilon_{ii} = \epsilon_{11} + \epsilon_{22} + \epsilon_{33}\ .$$

In addition, if thermal problems are being solved the current value for the temperature is passed as `td`. All material parameters for the current model are passed in the arrays `d(*)` and `ud(*)`. The array `d(*)` contains parameters assigned by standard *FEAP* commands as described in Tables 5.5 to 5.5 and the array `ud(*)` contains values as assigned in the user module `UMATIn`.

Other values for use in writing material models can be obtained from parameters in common blocks. For models which depend on position in the body the values of the reference and current coordinates for the constitutive point are passed in common block `elcoor` which contains the values in

```
        real (kind=8) :: xref   ,xcur
        common /elcoor/  xref(3),xcur(3)
```

For models that may need to use an incremental formulation with

$$\Delta\boldsymbol{\epsilon} = \boldsymbol{\epsilon}_{n+1} - \boldsymbol{\epsilon}_n$$

the array for strains may be dimensioned as `eps(9,2)` where the first 6 entries of `eps(9,1)` store the strains at t_{n+1} and the first 6 entries of `eps(9,2)` store those at t_n. The extra entries are not defined as they are provided only for use in the finite deformation form of the model described next.

Finite deformation models

For finite deformation models the deformation gradient is passed to `UMATLn` in the argument array `f(3,3,4)` where `f(3,3,1)` defines $\mathbf{F}_{n+1}$, `f(3,3,2)` defines $\mathbf{F}_n$, `f(3,3,3)` defines $\mathbf{G}_{n+1}$ and `f(3,3,4)` defines $\mathbf{G}_n$. The deformation gradient is stored as

$$\begin{aligned}
\text{f(i,J,1)} &= F_{iJ}(t_{n+1}) \\
\text{f(i,J,2)} &= F_{iJ}(t_n) \\
\text{f(i,J,3)} &= G_{iJ}(t_{n+1}) = F_{iJ}(t_{n+1}) - \delta_{iJ} \quad \text{and} \\
\text{f(i,J,4)} &= G_{iJ}(t_n) = F_{iJ}(t_n) - \delta_{iJ}
\end{aligned}$$

where G_{iJ} are displacement gradients. Stress and moduli are to be returned in the argument arrays dimensioned as `sig(6)` and `dd(6,6)`, respectively. Cauchy stresses and their moduli are returned using Voigt notation where stresses are ordered as

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{22} & \sigma_{33} & \sigma_{12} & \sigma_{23} & \sigma_{31} \end{bmatrix}^T$$

with corresponding order for the moduli. All real values are in double precision (i.e., `REAL*8`).

When `UMATLn` is called the model `n` will be that which is defined in the module `UMATIn`. Current values of the deformation gradient are, as mentioned above, passed in the array `f(3,3,4)` and the determinant of the deformation gradient in the parameter `theta(4)` where

$$\theta_1 = \det \mathbf{F}_{n+1} \quad \text{and} \quad \theta_2 = \det \mathbf{F}_n$$

```
      subroutine umatl1(eps,theta,td,d,ud,hn,h1,nh,ii,istrt,sig,dd,isw)

!-----[--.----+----.----+----.-----------------------------------------]
!     Purpose: User Constitutive Model

!     Input:
!          eps(*)  -  Current strains at point
!          theta   -  Trace of strain at point
!          td      -  Temperature change
!          d(*)    -  Program material parameters
!          ud(*)   -  User material parameters
!          hn(nh)  -  History terms at point: t_n
!          h1(nh)  -  History terms at point: t_n+1
!          nh      -  Number of history terms
!          ii      -  Current point number
!          istrt   -  Start state: 0 = elastic; 1 = last solution
!          isw     -  Solution option from element

!     Output:
!          sig(6)  -  Stresses at point.
!          dd(6,6) -  Current material tangent moduli

!-----[--.----+----.----+----.-----------------------------------------]
      implicit none

      integer       :: nh,ii,istrt, isw, i
      real (kind=8) :: td
      real (kind=8) :: eps(*),theta(*),d(*),ud(*),hn(nh),h1(nh)
      real (kind=8) :: sig(6),dd(6,6)

!     Dummy model:  sig = ud(1)*eps

      if(isw.eq.14) the ! Set initial values for history parameters

      !  .... None needed for this model

!     Compute tangent and stress

      else

        do i = 1,6
          dd(i,i) = ud(1)
          sig(i)  = ud(1)*eps(i)
        end do

      endif

      end subroutine umatl1
```

Figure 4.7: Sample UMATLn module for small deformation

In addition

$$\theta_3 = \det \mathbf{F}_{n+1} - 1 \quad \text{and} \quad \theta_4 = \det \mathbf{F}_n - 1$$

If thermal problems are being solved the current value for the temperature is passed as `td`. All material parameters for the current model are passed in the arrays `d(*)` and `ud(*)`. The array `d(*)` contains parameters assigned by standard *FEAP* commands as described in Tables 5.5 to 5.5 and the array `ud(*)` contains values as assigned in the user module `UMATIn`.

Other values for use in writing material models can be obtained from parameters in common blocks. For models which depend on position in the body the values of the reference and current coordinates for the constitutive point are passed in common block `elcoor` which contains the values in

```
real (kind=8) :: xref   ,xcur
common /elcoor/  xref(3),xcur(3)
```

Internal variable storage and use

For constitutive equations with additional (internal) variables that evolve in *time*, users must define entries for the `h1(*)` array. The number of entries available in the array for *each evaluation* (i.e., each quadrature point) is `nh`. The value for `nh` is defined by the parameter `n1` in module `UMATIn` (see Fig. 4.6). Values from the previous time step are passed back to the module in the array `hn(*)` (which also contains `nh` entries). Users should **never** modify entries in the `hn(*)` array. Finally, the values of the element operation switch is passed as the parameter `isw` (See Chapter 5 for operations performed during different values of `isw`).

In particular, when `isw=14` any non-zero values for internal variables must be set. It is not necessary to set zero values. Generally, for other values of `isw` and using the above information, users *must* compute values for the stress and the associated tangent matrix. These are returned to the element in the arrays `sig(6)` and `dd(6,6)`. In addition, updates for any of the history parameters must be assigned in the array `h1(*)` and returned to the element. Values of history variables returned are not used for all values of `isw` (e.g., when reporting or projecting stresses under `isw = 4` and `isw = 8` they are not saved). Values retained in the `h1(*)` array are copied to the `hn(*)` array each time the command statement `TIME` is issued in a solution.

```
      subroutine umatl1(f,detf,td,d,ud,hn,h1,nh,ii,istrt,sig,dd,isw)

!-----[--.----+----.----+----.-----------------------------------]
!     Purpose: User Constitutive Model

!     Input:
!          f(3,3,*)-  Deformation gradient (finite deformation)
!          detf(*) -  Determinant of deformation gradient
!          td      -  Temperature change
!          d(*)    -  Program material parameters
!          ud(*)   -  User material parameters
!          hn(nh)  -  History terms at point: t_n
!          h1(nh)  -  History terms at point: t_n+1
!          nh      -  Number of history terms
!          ii      -  Current point number
!          istrt   -  Start state: 0 = elastic; 1 = last solution
!          isw     -  Solution option from element

!     Output:
!          sig(6)  -  Stresses at point.
!          dd(6,6) -  Current material tangent moduli
!-----[--.----+----.----+----.-----------------------------------]
      implicit none

      integer       :: nh,istrt, ii, isw
      real (kind=8) :: td
      real (kind=8) :: f(3,3,*),detf(*),d(*),ud(*),hn(nh),h1(nh)
      real (kind=8) :: sig(6),dd(6,6)

!     Model:

      if(isw.eq.14) then  ! Set any initial values for history

      else                ! Compute model tangent and stress

        ...

      endif

      end subroutine umatl1
```

Figure 4.8: Sample UMATLn module for finite deformation

4.5.3 Accessing element and nodal data

In some user material models it is necessary to relate the behavior to additional data, such as an orientation field or other nodal related data. This may be accessed by retrieving the current element number from the common block `eldata`, which may be included in the `umatl` module using

```
include 'eldata.h'  ! n_el & ma values
```

and using the integer parameter '`n_el`' (which is the current element number being processed).[1] The value of the material model, '`ma`' begin processed is also available in the same include. With this information it is possible to access the element connection array `ix(nen1,numel)` using its pointer. This may be most easily obtained by adding statements of the form

```
include 'pointer.h' ! np(*) values
include 'comblk.h'  ! hr(*) & mr(*) arrays
      ....
call sub...(n_el,mr(np(33)), ..)  ! np(33) = ix pointer
```

and the subroutine using

```
subroutine sub...(n_el, ix,  ..)
implicit   none
include   'eldata.h' ! nel  value
include   'sdata.h'  ! nen1 value
integer (kind=4) :: ix(nen1,*)
      ...

do i = 1,nel
  node = ix(i,n_el)
      ...
```

Once the value of `node` is known other arrays may be used, for example nodal orientation arrays defined by the programmer as `umesh` modules, etc.

All *FEAP* arrays are similarly available in *any module* a programmer wants to develop.

[1]Earlier versions may use `n` as the element number.

4.5.4 Auto time step control

The solution command:

```
AUTO MATErial rvalu(1) rvalu(2) rvalu(3)
```

initiates an attempt to control the solution process by a variable time stepping algorithm based on a user set value in the material constitution. The value to be set is named `rmeas` which is passed between constitution and solution modules in the labeled common

```
real (kind=8) :: rmeas,rvalu
logical       ::               aratfl
common /elauto/  rmeas,rvalu(3),aratfl
```

The three parameters may be used in defining an acceptable value for `rmeas`. The algorithm coded monitors the solution during a standard iteration process set by, for example:

```
LOOP,,n
  TANG,,1
NEXT
```

If during any iteration up to `n` the value of `rmeas` exceeds a value of 2 (`rmeas = 0` at the start of the loop) a new value of Δt is immediately set to

$$\Delta t_{new} = 0.85\,\Delta t / rmeas$$

and the iteration process is started over. On the other hand if convergence occurs during the time step and the value of `rmeas` is smaller than 1.25, the time step is adjusted according to

$$\begin{array}{ll} \Delta t_{new} = 1.50\,\Delta t & ;\; rmeas \leq 0.5 \\ \Delta t_{new} = 1.25\,\Delta t & ;\; 0.5 < rmeas \leq 0.8 \\ \Delta t_{new} = \Delta t / rmeas & ;\; 0.8 < rmeas \end{array}$$

Finally, if convergence does not occur with in the `n` steps, then the time step is reset according to

$$\begin{array}{ll} \Delta t_{new} = 0.85\,\Delta t / rmeas & ;\; 1.25 < rmeas \\ \Delta t_{new} = \Delta t / 3 & ;\; \text{otherwise.} \end{array}$$

After any of the above adjustments the value of `rmeas` is reset to zero (0).

An optimal value of `rmeas` is 1.25 – which leaves the step unchanged. The above algorithm was proposed by Weber *et al.* [2].

4.5.5 Push forward routines

When developing constitutive models it is often necessary to push quantities forward from the reference configuration to the current configuration. For example, a vector V_I in the reference configuration can be pushed forward to the current configuration as

$$v_i = \frac{1}{J} F_{iI} V_I$$

In elements included in the program the deformation gradient is computed at the current time and the previous (converged) time; in addition the displacement gradient is also computed at the same times. The displacement gradient is expressed as

$$F_{iI} = \delta_{iI} + G_{iI}$$

where δ_{iI} is an identity tensor. A utility routine to push forward a vector is accessed using

```
       call pusht1(F, V, v, J, flag)
```

where `F(3,3)` is the *displacement gradient* if the logical parameter `flag` is true and is the *deformation gradient* if `flag` is false. The array `V(3)` passes the reference configuration vector and `v(3)` returns the current configuration vector. The parameter `J` may be passed as the determinant of the deformation gradient, or if no scaling is required as unity. The same routine may also be used to perform a *pull back* from the current configuration to the reference configuration by replacing the deformation gradient by its inverse.

The push forward of a second rank tensor is given by

$$a_{ij} = \frac{1}{J} F_{iI} A_{IJ} F_{jJ}$$

and may be implemented using the call

```
       call pusht2(F, A, a, J, flag)
```

where `F(3,3)` is the *displacement gradient* when the logical parameter `flag` is true and the *deformation gradient* when the logical parameter `flag` is false.

The above two routines work directly with the tensor components; however, routines also are provided that work in Voigt notation. For a symmetric second rank tensors the routine is

```
        call pushr2(f, S, s, J)
```

where `S(6)` is the reference configuration tensor ordered as

$$S_I = \begin{bmatrix} S_{11} & S_{22} & S_{33} & S_{12} & S_{23} & S_{31} \end{bmatrix}$$

and the current configuration `s(6)` in Voigt notation by

$$s_i = \begin{bmatrix} s_{11} & s_{22} & s_{33} & s_{12} & s_{23} & s_{31} \end{bmatrix}$$

The deformation gradient is `F(3,3)` and `J` is its determinant. Note that in some instances *FEAP* stores the deformation gradient as a 9-component vector ordered as

$$F_{iI} = \begin{bmatrix} F_{11} & F_{21} & F_{31} & F_{12} & F_{22} & F_{32} & F_{13} & F_{23} & F_{33} \end{bmatrix}$$

This ordering permits passing the array in either the 9-component `F(9)` form or in the two index `F(3,3)` form with identical result.

The push forward of fourth-order tensors (e.g., material moduli) is accomplished in Voigt notation as

$$\mathbf{d} = \frac{1}{J}\, \mathbf{T}_l^T \, \mathbf{D} \, \mathbf{T}_r$$

which in index form is given by

$$d_{ij} = \frac{1}{J} Tl_{ki} D_{kl} Tr_{lj}$$

here all arrays are of size 6 and for some algorithms the l and r indices are different. This form is used to replace the tensor form

$$c_{ijkl} = \frac{1}{J} F1_{iI} F2_{jJ} C_{IJKL} F3_{kK} F4_{lL}$$

Thus, it is necessary to first map the left side deformation gradient `F1` and `F2` onto the `Tl` matrix. This is accomplished using the Voigt ordering and implemented by calling the routine

$$Tl_{Nm} \leftarrow F1_{iI}\, F2_{jJ} \quad ; \quad N \text{ for } IJ \quad ; \quad n \text{ for } i,j$$

The above may be performed using

```
        call tranr4(F1, F2, Tl, flag)
```

where `F1` and `F2` are displacement gradients when `flag` is true otherwise *deformation gradients* when false. If the algorithm has all different descriptions for the various `Fi` then the routine may need to be called twice. Once the `Tl`, `Tr` are known the fourth rank tensor in Voigt notation is pushed forward using

```
        call pushr4(Tl, Tr, D, d, J)
```

in which all arrays are of size 6.

4.5.6 Polar decompositions

The right polar decomposition of the deformation gradient is given by the relation

$$F_{iJ} = R_{iI}\, U_{IJ} \tag{4.1}$$

where R_{iI} is a rotation and U_{IJ} the right stretch tensor. The rotation tensor satisfies the orthonormal relation

$$R_{iI}\, R_{iJ} = \delta_{IJ} \tag{4.2}$$

The computation of the stretch tensor may be obtained from the square root of the right Cauchy-Green tensor since

$$C_{IJ} = F_{iI}\, F_{iJ} = R_{iK} U_{KI}\, R_{iL} U_{LJ} = U_{KI} U_{KJ} \tag{4.3}$$

If the deformation gradient is written in terms of the displacement gradient as

$$F_{iI} = \delta_{iI} + G_{iI} \tag{4.4}$$

then

$$C_{IJ} - \delta_{IJ} = \delta_{iI} G_{iJ} + \delta_{iJ} G_{iI} + G_{iI} G_{iJ} = H_{IJ} \tag{4.5}$$

The result H_{IJ} may be written in the spectral form

$$H_{IJ} = \sum_{a=1}^{3} N_{aI} \lambda_a N_{aJ} \tag{4.6}$$

which may then be used to compute the stretch tensor as

$$U_{IJ} = \sum_{a=1}^{3} N_{aI} (1 + \lambda_a)^{1/2} N_{aJ} \tag{4.7}$$

and also its inverse as

$$U^{-1}_{IJ} = \sum_{a=1}^{3} N_{aI} (1 + \lambda_a)^{-1/2} N_{aJ} \tag{4.8}$$

The rotation tensor may then be obtained from

$$R_{iI} = F_{iJ}\, U^{-1}_{JI} = \delta_{iJ}\, U_{JI} + G_{iJ}\, U_{JI} \tag{4.9}$$

The left polar decomposition is expressed as

$$F_{iI} = V_{ij}\, R_{jI} \tag{4.10}$$

which has a similar solution as that described above.

Polar decomposition in *FEAP*

Users may include the right polar decomposition algorithm in modules programmed in Fortran by including including:

```
        call polar_ru( f, r, u, flag}
```

where `f(3,3)` is the deformation gradient when `flag=.true.`; and the displacement gradient if `flag=.false.`. The value of the rotation is returned in the array `r(3,3)` and the right stretch in `u(3,3)`.

The left polar decomposition may be obtained using

```
        call polar_vr( f, v, r, flag}
```

where `f(3,3)` is the deformation or displacement gradient depending on whether `flag` is true or false, respectively. The return values are the left stretch tensor `v(3,3)` and the rotation `r(3,3)`.

4.5.7 Numerical differentiation: Complex step

In developing constitutive modules for *FEAP* it is often quite easy to determine the expression for the stress (or derived variable) but may be more difficult to obtain the linearization for the algorithmic tangent matrix (moduli in the case of stress). For these situations a numerical method of differentiation becomes attractive and useful. Indeed, in *FEAP* it is possible to compute an estimate to the the *tangent stiffness matrix* using the command: `TANG NUMErical`. In order to use modules without modification the differentiation is done using a simple finite difference approximation for the first derivative of the residual vector.

As an alternative to derivatives in real arithmetic a very accurate computation may be obtained by performing the derivative in complex arithmetic. As a simple example consider the complex scalar function

$$f(z) = f(u + ih) \tag{4.11}$$

where $f(u)$ is a real function and h a parameter in the direction of u. A series expansion around the point u is given by

$$f(u + ih) = f(u) + ih\, f'(u) - \tfrac{1}{2!}\, h^2\, f''(u) - \tfrac{1}{3!}\, ih^3 f^{(3)}(u) \cdots \tag{4.12}$$

Taking the real part of the expansion gives

$$\Re f(u + ih) = f(u) - \tfrac{1}{2!}\, h^2\, f''(u) \cdots \tag{4.13}$$

Similarly taking the imaginary part gives

$$\Im f(u + ih) = h\, f'(u) - \tfrac{1}{3!}\, h^3 f^{(3)}(u) \cdots \tag{4.14}$$

Thus, assuming higher derivative behave smoothly, one may take a small value for h and obtain the derivative to numerical precision from

$$f'(u) = \Im f(u + ih)/h \tag{4.15}$$

Similarly, to numerical precision,

$$f(u) = \Re f(u + ih) \tag{4.16}$$

For the development of a constitutive relation or an element stiffness matrix, the scalar functions f and u may be replaced by vectors and the derivative carried out by perturbations on each component independently. This results in columns of the derivative (tangent array) for each perturbation. This method is called a *complex step* algorithm.[3, 4, 5, 6, 7, 8, 9, 10, 11, 12] Development of modules in Fortran are quite easy to incorporate complex arrays and variables using the declaration

```
        complex (kind=8) :: <list of variables>
```

In addition most intrinsic functions (e.g., `exp(z)`) may also be used with complex arguments `z`. This makes the complex step method very attractive for use in constructing the required tangents.

Chapter 5

ADDING ELEMENTS

FEAP permits users to add their own element modules to the program by writing a single subprogram called

```
subroutine elmtnn(d,ul,xl,ix,tl,s,r,ndf,ndm,nst,isw)
```

where `nn` may have values between `01` and `50` or for interface type elements by adding a single module called

```
subroutine intf0n(d,ul,xl,ix,tl,s,r,ndf,ndm,nst,isw)
```

where `n` has values between `2` and `5`. The basic steps for either form are identical. Copies of a required framework for all user elements are located in the `./user` directory. Each element module has the basic structure required for implementation of the steps described below. Most are set using an `if-then-else` form (as shown in Figure 5.1), however, `elmt11.f` is set using a `select-case` form (as shown in Figure 5.2) and may be substituted in the others if desired.

Build of executable

To add a user element:

- Copy the module `elmtnn.f` (where `nn` is between `01` and `50`) from the directory `./user` to the directory where the development will be made. *Do not delete the module in the* `./user` *directory. Do not change the name of the* `elmtnn.f` *routine or any of the variables in the argument list.*
- Build an archive (library module) of the program (including all the routines in the *./user* directory).

- Build the final program by combining the archive with: (a) the main program `feap86.f` which is located in the `./main` directory; and (b) the new element module `elmtnn.f` and any other new routines called by the element.

Hint: It is recommended that a temporary write be added as the first executable statement of the new element routine to ensure that the correct routine is accessed.

Structure of element

The basic structure for an element routine is shown in Figures 5.1 Part1 and Part 2.

Information is provided to the element subprogram through data passed as arguments and data passed in common blocks. The data passed as arguments consists of eleven (11) items which are briefly described in Table 5.1[1].

FEAP carries out tasks according to the parameter value, `ISW`, passed to the `ELMTnn` subprogram. A short description of the current task carried out by each value is given in Table 5.2.

To use basic solutions available in *FEAP* it is necessary to program tasks in Table 5.2

[1]Note in Table 5.1 that *FEAP* transfers the values for most of the solution parameters in array `UL(NDF,NEN,*)` at time t_{n+a}, where a denotes a value between 0 and 1. The value of a is 1 (i.e., values are reported for time t_{n+1}) unless generalized midpoint integration methods are used. For the present we will assume a is 1.

```
      subroutine elmtnn(d,ul,xl,ix,tl,s,r,ndf,ndm,nst,isw)

!     Prototype FEAP Element Routine:  nn = 01 to 50

      implicit none

!     Common blocks:  See Figure 5.2.
      integer       :: ndf,ndm,nst,isw
      integer       :: ix(*)
      real (kind=8) :: d(*),ul(ndf,*),xl(ndm,*),tl(*)
      real (kind=8) :: s(nst,nst),r(nst)

      if(isw. lt. 0) then
        utx(1) = 'Name_U_Want'  (Name of element type)
      elseif(isw.eq.0 .and. ior.lt.0) then
!       Return: Output of element description
        write(*,*) '   Elmt  1: Element description'
           ..
```

Figure 5.1: *FEAP* Element Subprogram. Part 1.

```
      elseif(isw.eq.1) then
!       Input/output of property data after command: 'mate'
!         d(*) stores information for each material set
!       Return: pstyp = <0,1,2,3> for dimension of mesh plots
!       Return: istv  = maxiumum number of element projections
!                       (default: project 8 quantities)
!       Return: nh1   = number of nh1/nh2 words/element
!       Return: nh3   = number of nh3     words/element

      elseif(isw.eq.2) then
!       Check element for errors.  Negative jacobian, etc.

      elseif(isw.eq.3) then
!       Return: Element coefficient matrix and residual
!         s(nst,nst) element coefficient matrix
!         r(ndf,nen) element residual
!         hr(nh1)    history data base: previous time step
!         hr(nh2)    history data base: current  time step
!         hr(nh3)    history data base: time independent

      elseif(isw.eq.4) then
!       Return: Output element quantities (e.g., stresses)

      elseif(isw.eq.5) then
!       Return: Element mass matrix       (imtype = 1)
!       Return: Element geometric tangent (imtype = 2)
!         s(nst,nst) consistent matrix/geometric tangent
!         r(ndf,nen) diagonal matrix

      elseif(isw.eq.6) then
!       Return: Residual only
!         r(ndf,nen) element residual

      elseif(isw.eq.7) then
!       Return: Surface loading for element
!         s(nst,nst) coefficient matrix
!         r(ndf,nst) nodal forces

      elseif(isw.eq.8) then
!       Return: Element projections to nodes (diagonal)
!         p(nen)   projection weight: wt(nen)
!         s(nen,*) projection values: st(nen,*)
!       Return: iste = number of projections
      endif
      end subroutine elmtnn
```

Figure 5.1: *FEAP* Element Subprogram. Part 2.

labeled as R. Elements with local variables that need to be retained between subsequent time steps (*history variables*) are defined as described in Section 5.7. In this case it may be necessary to code Task 12 for any variable transformation. Task 14 is used to set non-zero initial values of history variables (zero values are set by default). Finally, if special plotting options are desired it may be necessary to program Task 20 (contours for element variables such as stress, strain, etc. are computed in Task 8).

It is not necessary to implement optional tasks in an element, however, for those tasks that are not implemented it is important that the element routine not perform any calculations. Thus if the form of the branch is programmed as an `IF-THEN-ELSE` construct as shown in Fig. 5.1 then the `ELSE` should not carry out any operations *unless all options for `ISW` are programmed.* Similarly if the element is programmed using a `SELECT-CASE` form shown in Figure 5.2 the `CASE DEFAULT` should not perform any operations.

Parameter	Description
d(*)	Element data parameters
	(Moduli, body loads, etc.)
ul(ndf,nen,j)	Element nodal solution parameters
	nen is number of nodes on an element (max)
	j = 1: Displacement $u_{n+a}^{(k)}$
	j = 2: Increment $u_{n+a}^{(k)} - u_n$
	j = 3: Increment $u_{n+1}^{(k)} - u_{n+1}^{(k-1)}$
	j = 4: Rate $v_{n+a}^{(k)}$
	j = 5: Rate $a_{n+a}^{(k)}$
	j = 6: Rate v_n
xl(ndm,nen)	Element nodal reference coordinates
ix(nen)	Element global node numbers
tl(nen)	Element nodal temperature values
s(nst,nst)	Element matrix (e.g., stiffness, mass)
r(ndf,nen)	Element vector (e.g., residual, mass)
	may also be used as r(nst)
ndf	Number unknowns (max) per node
ndm	Space dimension of mesh
nst	Size of element arrays S and R
	N.B. Normally nst = ndf*nen
isw	Task parameter to control computation
	See prototype element in Figure 5.1

Table 5.1: Arguments of *FEAP* Element Subprogram.

`isw` Task	Type	Description	Access Command	Calling Program
-1	O	Set name in `utx(1)`	Called by default	`pcontr`
0	O	Output label	`SHOW ELEM`	`pmacr5`
1	R	Input `d(*)` parameters	Mesh:`MATE,n`	`pmatin`
2	O	Check elements	Soln:`CHECk`	`pform`
3	R	Compute tangent/residual	Soln:`TANG`	`pform`
		Store in `S/r`	`UTAN`	`pform`
4	O	Output element variables	Soln:`STRE`	`pform`
5	E	Compute cons/lump mass	Soln:`MASS`	`pform`, `formrb`
		Store in `S/r`	`MASS,LUMP`	`pform`
6	R	Compute residual	Soln:`FORM`,`REAC`	`pform`
			Plot:`REAC`	`pform`
7	O	Surface load/tangents	Mesh:`SLOAd`	`ploadl`
8	O	Nodal projections	Soln:`STRE NODE`	`pform`
			Plot:`STRE`,`PSTR`	`pform`
9	O	Damping	Soln:`DAMP`	`pform`
10	O	Augmented Lagrangian update	Soln:`AUGM`	`pform`
11	O	Error estimator	Soln:`ERRO`	`pform`
12	R	History update. For special treatments else return	Soln:`TIME`	`pform`
13	O	Energy/momentum	Soln:`TPLO`,`ENER`	`pform`
14	R	Initialize history	`BATCh`,`INTEr`	`pform`
15	O	Body force	Mesh:`BODY`	`pform`
16	O	J integrals	Soln: `JINT`	`pform`
17	O	Set after activation	Soln:`ACTI`	`pform`
18	O	Set after deactivation	Soln:`DEAC`	`pform`
19		NOT AVAILABLE: used in modal/base. Uses `isw = 5`	Soln:`BASE`	`pform`
20	O	Element plotting	Plot:`PELE`	`pform`
21	O	Critical time step calculation	Soln:`TIME EXPL`	`pform`
22	O	stress/strain volume average	Soln:`STRE AVER`	`pform`
23	O	Compute element loads only	Soln:`ARCL`	`pform`
25	O	Zienkiewicz-Zhu projection	Soln:`ZZHU`	`pform`
26	R	Used to compute mesh boundary	Called by default.	`pextndc`

Table 5.2: Task Options for *FEAP* Element Subprogram. R = Required; O = Optional; E = For eigensolutions

N.B. Finally, the old form

```
       select case (isw)
         case(-1)
           utx(1) = 'Name_U_Want'
         case(1)
!          Input material parameters
           ...
           pstyp = <0,1,2,3>   ! Dimension of mesh plot
           istv  = max(istv,*> ! * = max no. element projections
         case default
           ...
       end select
```

Figure 5.2: *FEAP* Element Subprogram. Case form.

```
       go to (1,2,..... ), isw
       return
!      Input Material Properties
 1     ..... etc.
```

is not recommended, however, if it is used the `RETURN` statement should *always be included* as shown. This prevents any unexpected execution of a statement that appears after the `GO TO`.

Some of the options for additional data passed through common blocks is shown in Figure 5.3 with each variable defined in Table 5.3. Also, in Figure 5.4 the reference to common blocks using include statements is shown. In the prototype routine the number of nodes on an element (`nen`) which is used to dimension `ul` is passed in the labeled common `/cdata/`. Additional discussion is given below on use of some of the other data passed through the common blocks.

5.1 Material property storage

The material parameters to be stored in the array `D` with pointer `np(25)` may be input using the subprogram `INMATE`. This subroutine is accessed by the statement:

```
       call inmate(d,tdof, nev, type)
```

where `d` is the array storing the material parameters; `tdof` is returned as the parameter to access temperature; `nev` is the number of element history variables to allocate to `nh1`; and `type` is an input to define the element type (the various `type` of elements allowed is specified in the module `inmate.f`)..

```
character (len=4) :: o,head
common /bdata/       o,head(20)

integer      :: numnp,numel,nummat,nen,neq,ipr, netyp, cnel
common /cdata/  numnp,numel,nummat,nen,neq,ipr, netyp, cnel

integer      :: nstep,niter,nform,naugm, titer,taugm,tform
common /counts/ nstep,niter,nform,naugm, titer,taugm,tform

integer      :: iaugm,iform,intvc,iautl, nstepa, nsplt
common /counts/ iaugm,iform,intvc,iautl, nstepa, nsplt

character (len=17) :: ecapt    , dcapt
common    /elcapt/    ecapt(50), dcapt(50)

real (kind=8) ::dm
integer       ::   n_el,ma,mct,iel,nel,pstyp,eltyp,eltyp2,eltyp3
common /eldata/ dm,n_el,ma,mct,iel,nel,pstyp,eltyp,eltyp2,eltyp3

real (kind=8) ::tt
common /elplot/ tt(1000)

real (kind=8) ::bpr,   ctan,   psil
common /eltran/ bpr(3),ctan(3),psil

real (kind=8) ::ut
common /eluser/ ut(1000)

integer      :: nh1,nh2,nh3,ht1,ht2,ht3  ! int*4 or int*8
common /hdata/  nh1,nh2,nh3,ht1,ht2,ht3

integer      :: nlm,plm,nge,pge          ! int*4 or int*8
common /hdata/  nlm,plm,nge,pge
```

Figure 5.3: Partial list of *FEAP* element common blocks. (N.B. All variables may not be included above.): Part 1

This routine inputs the commands as described in the user manual and stores the data for each material set into the `D` array elements as described in Table 5.5. Users should always verify that table list is correct by checks to module `inmate` located in the `./elements/material` directory.

5.2 Element matrix dimensions

Each element has the capability to form two arrays: a matrix, `S`, and a vector, `R`. For example, when `isw = 3` the matrix stores the problem *tangent array* and the vector

```
integer       :: ior,iow,ilg
common /iofile/ ior,iow,ilg

logical       :: keepfl,wprt
common /iofile/ keepfl,wprt

integer       :: nph,ner                  ! int*4 or int*8
real (kind=8)::          erav,jshft
common /prstrs/ nph,ner,erav,jshft

integer       :: ndf,ndm,nen1,nst,nneq,ndl,nnlm,nadd
common /sdata/  ndf,ndm,nen1,nst,nneq,ndl,nnlm,nadd

real (kind=8):: ttim,dt,c1,c2,c3,c4,c5, chi, dtcr
integer       ::                                 idyn0
common /tdata/  ttim,dt,c1,c2,c3,c4,c5, chi, dtcr, idyn0

integer (kind=8) :: np     ,up
common /pointer/    np(400),up(200)

real (kind=8):: hr
integer       ::          mr
common /comblk/ hr(1024),mr(1024)
```

Figure 5.3: Partial list of *FEAP* element common blocks: Part 2

the problem *residual array*. When `isw = 5` the matrix stores the `consistent mass array` and the vector a `lumped mass array`.

In *FEAP* the element tangent matrix, $\mathbf{S}_{ij}$, is stored as a two dimensional array which is dimensioned as `s(nst,nst)`, where `nst` is the product of `ndf` and `nen` plus any element and global equations, with `ndf` the *maximum number of degree-of-freedoms*

```
include  'bdata.h'
include  'cdata.h'
include  'counts.h'
include  'eldata.h'
include  'elplot.h'
include  'eltran.h'
include  'hdata.h'
include  'iofile.h'
include  'prstrs.h'
include  'tdata.h'
include  'pointer.h'
include  'comblk.h'
```

Figure 5.4: *FEAP* Element Common Blocks using Includes.

Common Name	Variable	Definition
`bdata`	`o`	Page eject option
	`head`	Title record
`cdata`	`numnp`	Number of mesh nodes
	`numel`	Number of mesh elements
	`nummat`	Number of material sets
	`nen`	Maximum nodes/element
	`neq`	Number active equations
	`ipr`	Real variable precision
`counts`	`nstep`	Total number of time steps
	`niter`	Number of iterations current step
	`naugm`	Number of augments current step
	`titer`	Total iterations
	`taubm`	Total augments
	`iaugm`	Augmenting counter
	`iform`	Number residuals in line search
`elcapt`	`dm`	Nodal & element plot captions
	`dcapt`	Nodal contour plot captions
	`ecapt`	Element contour plot captions
`eldata`	`dm`	Element proportional load
	`n_el`	Current element number
	`ma`	Current element material set
	`mct`	Print counter
	`iel`	User element number
	`nel`	Number nodes on current element
`elplot`	`tt`	Element stress values for `TPLOt`
`eltran`	`bpr`	Principal stretch
	`ctan`	Element multipliers
`eluser`	`ut`	Element user values for `TPLOt`

Table 5.3: *FEAP* common block partial list of definitions.

at any node in the problem and **nen** the maximum number of nodes on any element. The ordering of the unknowns into the first **ndf*nen** entries of **nst** must be carefully aligned in order for FEAP to properly assemble each element matrix into the global tangent. Element equations follow these and then finally any global equations (See Fig. 5.5). The ordering of the first row and column blocks is such that sub-matrices

Common Name	Variable	Definition
`hdata`	`nh1`	Pointer to t_n history data
	`nh2`	Pointer to t_{n+1} history data
	`nh3`	Pointer to element history
	`nlm`	Number of element equations
	`plm`	Partition of element equations
	`nge`	Number of global equations
	`pge`	Partition of global equations
`iofile`	`ior`	Current input logical unit
	`iow`	Current output logical unit
`prstrs`	`nph`	Pointer to global projection arrays
	`ner`	Pointer to global error indicator
	`erav`	Element error value
`sdata`	`ndf`	Maximum dof/node
	`ndm`	Mesh space dimension
	`nen1`	Dimension 1 on IX array
	`nst`	Size of element matrix
	`nneq`	Total dof in problem
`tdata`	`ttim`	Current time
	`dt`	Current time increment
	`ci`	Integration parameters
`comblk`	`hr`	Real array data
	`mr`	Integer array data

Table 5.4: *FEAP* common block partial list of definitions.

are defined for each node attached to the element. Thus

$$\mathbf{S} = \begin{bmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} & \mathbf{S}_{13} & .. \\ \mathbf{S}_{21} & \mathbf{S}_{22} & \mathbf{S}_{23} & .. \\ \mathbf{S}_{31} & \mathbf{S}_{32} & \mathbf{S}_{33} & .. \\ .. & .. & .. & .. \end{bmatrix}$$

where $\mathbf{S}_{ij}$ is the sub-matrix for nodal pairs i, j. Each of the sub-matrices is a square matrix of the size of the maximum number of degree-of-freedoms in the problem which is passed to the subprogram as `ndf`. Thus,

$$\mathbf{S}_{ij} = \begin{bmatrix} S^{ij}_{11} & S^{ij}_{12} & S^{ij}_{13} & .. \\ S^{ij}_{21} & S^{ij}_{22} & S^{ij}_{23} & .. \\ S^{ij}_{31} & S^{ij}_{32} & S^{ij}_{33} & .. \\ .. & .. & .. & S^{ij}_{ndf,ndf} \end{bmatrix}$$

in which S^{ij}_{ab} is an array coefficient for nodal pair i, j for the degree-of-freedom pair a, b.

Parameter	Name	Description
1	E	Young's modulus
2	ν	Poisson ratio
3	α	Thermal expansion coefficient
4	ρ	Mass density
5	-	Quadrature order for arrays & output
7	a	Mass interpolation ($a = 0$: Diagonal; $a = 1$: Consistent
8	rho_i	Rotational mass factor (plates/shells)
9	T_0	Stress free reference temperature
10	q	Loading intensity (plates/shells)
11	b_1	Body force/volume in 1-directions
12	b_2	Body force/volume in 2-directions
13	b_3	Body force/volume in 3-directions
14	h	Thickness (plates/shells)
15	`nh1`	History variable counter
16	`stype`	Two dimensional type: 1 - plane stress; 2 - plane strain; 3 - axisymmetric; 8 - axisymmetric-torsion; 9 - spherical[2]
17	`etype`	Element formulation: 1 - displ; 2 - mixed; 3 - enhanced; 7 - Uniform defm.; 8 - stabilized; 9 - incompressible.
18	`dtype`	Deformation type: <: finite; small >
19	`tdof`	Thermal degree-of-freedom
20	`imat`	Non-linear elastic material type
21	$d_{11},\ a_1$	Material elastic moduli, Fung parameter
22	$d_{22},\ a_2$	Material elastic moduli, Fung parameter
23	$d_{33},\ a_3$	Material elastic moduli, Fung parameter
24	$d_{12},\ a_4$	Material elastic moduli, Fung parameter
25	$d_{23},\ a_5$	Material elastic moduli, Fung parameter
26	$d_{31},\ a_6$	Material elastic moduli, Fung parameter
27	$g_{12},\ a_7$	Material elastic moduli, Fung parameter
28	$g_{23},\ a_8$	Material elastic moduli, Fung parameter
29	$g_{31},\ a_9$	Material elastic moduli, Fung parameter
30	C	Fung pseudo elastic model modulus

Table 5.5: Material Parameters.

Parameter	Name	Description
31	ψ	Orthotropic angle x_1 principal axis 1
32	A	Area cross section (beam/truss)
33	I_{11}	Inertia cross section (beam/truss)
34	I_{22}	Inertia cross section (beam/truss)
35	I_{12}	Inertia cross section (beam/truss)
36	J	Polar inertia cross section (beam/truss)
37	κ_1	Shear factor (plates/shells/beams)
38	κ_2	Shear factor plate
39	-	Non-linear flag (beam/truss)
40	-	Inelastic material model type
41	Y_0	Initial yield stress (Mises)
42	Y_∞	Final yield stress (Mises)
43	β	Exponential hardening rate
44	H_{iso}	Isotropic hardening modulus (linear)
45	H_{kin}	Kinematic hardening modulus (linear)
46	-	Yield flag
47	β_1	Orthotropic thermal stress
48	β_2	Orthotropic thermal stress
49	β_3	Orthotropic thermal stress
50	-	Error estimator parameter
51	ν_1	Viscoelastic shear parameter
52	τ_1	Viscoelastic relaxation time
53	ν_2	Viscoelastic shear parameter
54	τ_2	Viscoelastic relaxation time
55	ν_3	Viscoelastic shear parameter
56	τ_3	Viscoelastic relaxation time
57	`nvis`	Number of viscoelastic terms (1-3)
58	-	Damage limit
59	-	Damage rate
60	k	Penalty parameter

Table 5.5: (Cont.) Material Parameters.

Parameter	Name	Description
61	K_1	Fourier thermal conductivity
62	K_2	Fourier thermal conductivity
63	K_3	Fourier thermal conductivity
64	c	Fourier specific heat
65	ω	Angular velocity
66	Q	Body heat
67	-	Heat constitution added indicator
68	-	Follower loading indicator
69	-	Frame distributed load (framf3e.f only)
70	-	Damping factor
71	g_1	Ground acceleration factor
72	g_2	Ground acceleration factor
73	g_3	Ground acceleration factor
74	p_1	Ground acceleration proportional load number
75	p_2	Ground acceleration proportional load number
76	p_3	Ground acceleration proportional load number
77	a_0	Rayleigh damping mass ratio
78	a_1	Rayleigh damping stiffness ratio
79	-	Plate/Shell/Rod shear activation flag
80		Method: Type 1
81		Method: Type 2
82	-	Truss/Rod quadrature number
83	-	Axial loading value
84	-	Constitutive start indicator
85	-	Polar angle indicator
86	-	Polar angle coord_1
87	-	Polar angle coord_2
88	-	Polar angle coord_3
89	-	Constitution transient type
90	d_{31}	Plane stress recovery
91	d_{32}	Plane stress recovery
92	α_3	Plane stress recovery

Table 5.5: (Cont.) Material Parameters.

Parameter	Name	Description
93	`sref`	Shear center type
94	y_1	Shear center coordinate
95	y_2	Shear center coordinate
96	`lref`	Reference vector type
97	n_1	Reference vector parameter
98	n_2	Reference vector parameter
99	n_3	Reference vector parameter
100	-	Cross section shape type: 1 = rectangles; 2 = tube; 3 = Wide flange; 4 = Channel; 5 = Angle; 5 = Circle
101-126	-	Shape data
127	-	Surface convection (h)
128	-	Free-stream temperature (T_∞)
129	-	Reference absolute temperature
130	`nseg`	Number of hardening segments
131-148	-	Segment data sets e_p, Y_{iso}, H_{kin}
149	-	Total variables on frame section
150	-	Plastic kinematic hardening
151-156	-	Hardening: h_1, h_2, h_3, h_4, h_5, j_1
157	$\bar{F}$	Traction RVE constraints.
158	$\bar{Q}$	Thermal flux RVE constraints.
159	ngm	Number of RVE constraints
160	-	Initial stress flag
161-166	σ_{ij}	Initial stresses (constant)
167	-	Tension/compression only indicator
168	-	Thermal activation indicator
169	-	Mechanical activation indicator
170	-	Volume model number (default 1)
171	-	Plot projections on/off
172	`nvpr`	Number viscoelastic pressure terms (1-3)
173	μ_1	Viscoelastic volume/pressure parameter
174	τ_1	Viscoelastic relaxation time
175	μ_2	Viscoelastic volume/pressure parameter
176	τ_2	Viscoelastic relaxation time
177	μ_3	Viscoelastic volume/pressure parameter
178	τ_3	Viscoelastic relaxation time
179	-	Unused
180-181	-	Viscoplastic rate parameters
182	-	Nodal quadrature parameters
183	β_m	$\mathbf{M}_L - \mathbf{M}_C$ mass scaling factor
184	c	Estimate on maximum wave speed

Table 5.5: (Cont.) Material Parameters.

Parameter	Name	Description
185	-	Augmentation switch: <on/off>
186	-	Augmentation explicit indicator
187		Implicit = 0; Explicit = 1 element integration
188	-	Number stress components in rod elements
189	-	Nurbs & VEM flag
190-192	-	Nurbs quadrature values/direction
193	*tmat*	Thermal material numbers
194	ietype	Element type
195	$T - frac$	Fraction of work to heat
196	$q - prop$	Proportional load factor for pressure loading
197-198	-	Body patch loading values
199	-	Axisymmetric 1-d: Plane stress in thickness
200	*nsiz*	Size of modulus or compliance array
201-236	-	Anisotropic Modulus or Compliance array
237	-	Number of element global equations (`nge`
238	-	Partition of element global equations
239	-	Unused
240	-	0 = Element based; 1 = nodal based formulation
241	-	Number of active element degrees of freedom
242-248	V_1,V_2	Plastic Vector orientation
249-255	-	Reference vector types and values
260-279	nstv	Number structure vectors/values
280-282	g_i	Thermal-elastic temperature function
283	-	Unused
284-286	-	Delete element data
287	-	Total energy computation switch
288	-	Shell thickness change flag
289	-	Rate switch (on=0,off=1)
290-293	-	Constitutive equation coordinate frame
294	-	Rotatory inertia on/off flag
295-296	-	Body force *user* parameters
297	-	RVE type: 1 = Hill-Mandel; 2 = Irving-Kirkwood

Table 5.5: (Cont.) Material Parameters.

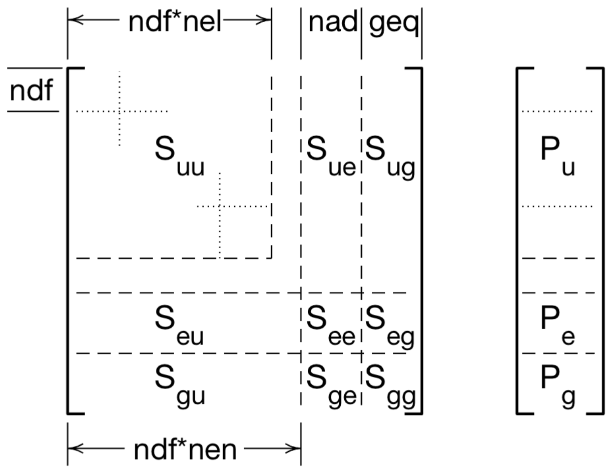


Figure 5.5: Tangent matrix and residual including element and global constraints.

In *FEAP*, the element residual may be stored as a one dimensional array which is dimensioned `r(nst)` with entries stored in the same order as the rows of the element tangent matrix or as a two dimensional array which is dimensioned as `r(ndf,nen)`. The one dimensional form of the residual is given as

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \\ \mathbf{R}_3 \\ \vdots \end{bmatrix}$$

where the entries in each submatrix are given as

$$\mathbf{R}_i = \begin{bmatrix} R_1^i \\ R_2^i \\ \vdots \\ R_{ndf}^i \end{bmatrix} .$$

The two dimensional form `r(ndf,nen)` places the entries $\mathbf{R}_i$ as columns. Accordingly,

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 & \mathbf{R}_3 & \cdots \end{bmatrix} .$$

The two forms for defining the residual `r` are equivalent based on the Fortran ordering of information into double subscript arrays.

If `ndf` is larger than needed for the element and residual the unused positions need not be defined (the tangent array `s` and the residual `r` are set to zero before each element routine is called).

The arrays `xl(i,j)`, `ul(i,j,1)`, `ul(i,j,4)` and `ul(i,j,5)` (described in Table 5.1) are used to obtain the nodal coordinates, displacements, velocities and accelerations, respectively.

When programming an element it is the users responsibility to decide the meaning for each degree-of-freedom. In all the standard elements provided with *FEAP* the degree-of-freedoms for displacements are assigned to the first `ndm` positions (where `ndm` is the spatial dimension of the mesh). In thermo-mechanical problems the thermal degree-of-freedom is normally located at `NDM+1`. The actual location of element degree-of-freedoms to the global degree-of-freedoms can be set in the input file by the data statements

```
MATErial ma
  etype  uel eset g_1 g_2 ... g_ndf
```

When `etype = user` the parameter `uel` defines the user element number. The `eset` parameter defines the values set for each element (by default it is `ma`). Finally the `g_i` values define the global degree-of-freedom for the "`i`" local degree-of-freedom. By default `g_i = i`. Thus, if the programmer is coding `ELMT02` and has placed the values for a scalar degree-of-freedom in the first degree-of-freedom in `S` and `R` it may be moved to global degree-of-freedom 4 using the input statements

```
MATErial ma
  USER   2  ,, 4
    ...
```

To assemble the element stiffness matrix it is often useful to define an integer indexing array, `sa(nen)` which may be set in Fortran using the statements[3]:

```
sa(1) = 0
do i = 2,nel
  sa(i) = sa(i-1) + ndf
end do ! i
```

The entries in the first `nd` degree-of-freedoms in the element matrix and vector may then be assembled using the statements

[3]If the include `qudshp.h` is used in the element the array is automatically defined and available.

```
do j = 1,nel      ! Column node loop
  do a = 1,nd     ! DOF loop
      ! 1-d r-array form
    r(sa(j)+a) = r(sa(j)+a) + ...
      ! 2-d r-array form
    r(a,j) = r(a,j) + ...
  end do ! a
  do i = 1,nel    ! Row node loop
    do b = 1,nd   ! DOF loops
      do a = 1,nd
        s(sa(i)+a,sa(j)+b) = s(sa(i)+a,sa(j)+b) + ...
           ...
```

This form ensures that the submatrices are properly aligned in the `s`-array and `r`-array.

5.3 Elements with internal equations

In some formulations it is convenient to use *non-nodal* degrees of freedom that are independent of a node. These may be of the Lagrange multiplier type or simply any other variable. To activate these element variables the include

```
include  'hdata.h'
```

must be available and the statement

```
nlm = var1
```

defined in the sections where `isw=1` occurs. The value of `var1` defines the number of element variables. If the equations are only active in one partition then the statement

```
plm = var2
```

is also given where `var2` defines the partition for element variables. If `plm` is zero the element equations are active in all partitions. The equations associated with these equations is located at the rows and columns greater than `nen*ndf` in the residual and tangent matrices. That is in rows and columns `nen*ndf+1` to `nen*ndf+var1`. See Sect. 5.5.1 for further details on setting element equations.

5.4 Non-linear Transient Solution Forms

Before describing the steps in developing an element we summarize first the basic structure of the algorithms employed by *FEAP* to solve problems. Each problem to be solved using an `ELMTnn` routine is established in a standard finite element form as described in standard references (e.g., *The Finite Element Method*, 4th ed., by O.C. Zienkiewicz and R.L. Taylor, McGraw-Hill, London, 1989 (vol 1), 1991 (vol 2)). Here it is assumed this step leads to a set of non-linear ordinary differential equations expressed in terms of nodal displacements, velocities, and accelerations given by $\mathbf{u}_i(t)$, $\dot{\mathbf{u}}_i(t)$, and $\ddot{\mathbf{u}}_i(t)$, respectively. We denote the differential equation for node-i as the residual equation:

$$\mathbf{R}_i(\mathbf{u}_i(t), \dot{\mathbf{u}}_i(t), \ddot{\mathbf{u}}_i(t), t) \;=\; \mathbf{0}\,.$$

To solve for the nodal displacements,velocities and accelerations it is necessary to introduce an algorithm to integrate the nodal quantities in time, specify a constitutive relation, and develop an algorithm to solve a (possibly) non-linear problem.

In *FEAP*, the integration method for nodal quantities is taken as a one step algorithm with each quantity defined only at discrete times t_n. Accordingly, we have displacements $\mathbf{u}_i(t_n)$ with velocities and accelerations denoted as

$$\dot{\mathbf{u}}_i(t_n) \;\approx\; \mathbf{v}_i(t_n)$$

and

$$\ddot{\mathbf{u}}_i(t_n) \;\approx\; \mathbf{a}_i(t_n)$$

A typical example for an integration algorithm for these discrete quantities is Newmark's method where

$$\mathbf{u}_i(\,t_{n+1}\,) \;=\; \mathbf{u}_i(\,t_n\,) \;+\; \Delta t\,\mathbf{v}_i(\,t_n\,) \;+\; \Delta t^2\,[(\,\frac{1}{2} \,-\, \beta\,)\,\mathbf{a}_i(\,t_n\,) \,+\, \beta\,\mathbf{a}_i(\,t_{n+1}\,)]$$

and

$$\mathbf{v}_i(\,t_{n+1}\,) \;=\; \mathbf{v}_i(\,t_n\,) \;+\; \Delta t\,[(\,1 \,-\, \gamma\,)\,\mathbf{a}_i(\,t_n\,) \,+\, \gamma\,\mathbf{a}_i(\,t_{n+1}\,)]$$

with $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{a}$ being the set of displacements, velocities, and accelerations at node-i, respectively.

A Newton method is commonly adopted to solve a non-linear (or linear) problem. To implement a Newton method it is necessary to linearize the residual equation. For *FEAP*, the Newton equation may be written as

$$\mathbf{R}_i^{(k+1)} \;=\; \mathbf{R}_i^{(k)} \;+\; \frac{\partial \mathbf{R}_i}{\partial \boldsymbol{\alpha}_j}\,|^{(k)}\; d\boldsymbol{\alpha}_j^{(k)} \;=\; \mathbf{0}$$

where $\boldsymbol{\alpha}_j$ is one of the variables at time t_{n+1} (e.g., $\mathbf{u}_j(\,t_{n+1}\,)$). We define

$$\mathbf{S}_{ij}^{(k)} \;=\; -\frac{\partial \mathbf{R}_i}{\partial \boldsymbol{\alpha}_j}\,|^{(k)}$$

and solve

$$\mathbf{S}_{ij}^{(k)}\,d\boldsymbol{\alpha}_j^{(k)} \;=\; \mathbf{R}_i^{(k)}\;.$$

The solution is updated using

$$\boldsymbol{\alpha}_j^{(k+1)} \;=\; \boldsymbol{\alpha}_j^{(k)} \,+\, d\boldsymbol{\alpha}_j^{(k)}\;.$$

In the above (k) is the iteration number for the Newton algorithm. To start the solution for each step, FEAP sets

$$\boldsymbol{\alpha}_j^{(0)}(\,t_{n+1}\,) \;=\; \boldsymbol{\alpha}_j(\,t_n\,)$$

where a quantity without the (k) superscript represents a converged value. For a linear problem, Newton's method converges in one iteration. Computing the residual after one iteration *must yield a zero value* to within the roundoff of the computer used. For non-linear problems, a properly implemented Newton's method *must exhibit a quadratic asymptotic rate of convergence.* Failure of the above performance for linear and non-linear cases implies a programming error in an implementation or lack of a consistently linearized algorithm (i.e., $\mathbf{S}_{ij}$ is not an exact derivative of the residual).

In a non-linear problem, Newmark's method may be parameterized in terms of increments of displacement, velocity, or acceleration. From the Newmark formulas, the relations

$$d\mathbf{u}_i \;=\; \beta\,\Delta t^2\,d\mathbf{a}_i$$

and

$$d\mathbf{v}_i \;=\; \gamma\,\Delta t\,d\mathbf{a}_i$$

define the relationships between the increments. Note that only scalar multipliers involving β, γ, and Δt are involved between the different measures.

The tangent matrix for the transient problem using Newmark's method may be expressed in terms of the incremental displacement, velocity, or acceleration. As an example, consider the case where the solution is parameterized in terms of increments of the displacements (i.e., $\boldsymbol{\alpha}_j$ is the displacement vector $\mathbf{u}_j$). For this case, the tangent matrix is (we do not show dependence on the iteration (k) for simplicity of notation)

$$\mathbf{S}_{ij}\,d\mathbf{u}_j \;=\; -\frac{\partial \mathbf{R}_i}{\partial \mathbf{u}_j}\,d\mathbf{u}_j \,-\, \frac{\partial \mathbf{R}_i}{\partial \mathbf{v}_k}\,\frac{\partial \mathbf{v}_k}{\partial \mathbf{u}_j}\,d\mathbf{u}_j \,-\, \frac{\partial \mathbf{R}_i}{\partial \mathbf{a}_k}\,\frac{\partial \mathbf{a}_k}{\partial \mathbf{u}_j}\,d\mathbf{u}_j\;.$$

Note that from the Newmark formulas

$$\frac{\partial \mathbf{a}_k}{\partial \mathbf{u}_j} = \frac{1}{\beta\,\Delta t^2}\boldsymbol{\delta}_{kj} \quad ; \quad \frac{\partial \mathbf{v}_k}{\partial \mathbf{u}_j} = \frac{\partial \mathbf{v}_k}{\partial \mathbf{a}_l}\frac{\partial \mathbf{a}_l}{\partial \mathbf{u}_j} = \frac{\gamma}{\beta\,\Delta t}\boldsymbol{\delta}_{kj}$$

in which $\boldsymbol{\delta}_{kj}$ is the Kronnecker delta identity matrix for the k,j nodal pair . From the residual we observe that

$$\mathbf{K}_{ij} = -\frac{\partial \mathbf{R}_i}{\partial \mathbf{u}_j} \quad ; \quad \mathbf{C}_{ij} = -\frac{\partial \mathbf{R}_i}{\partial \mathbf{v}_j} \quad ; \quad \mathbf{M}_{ij} = -\frac{\partial \mathbf{R}_i}{\partial \mathbf{a}_j}$$

define the tangent stiffness, damping, and mass, respectively. Thus, for the Newmark algorithm the total tangent matrix in terms of the incremental displacements is

$$\mathbf{S}_{ij} = \mathbf{K}_{ij} + \frac{\gamma}{\beta\,\Delta t}\mathbf{C}_{ij} + \frac{1}{\beta\,\Delta t^2}\mathbf{M}_{ij}\ .$$

For other choices of increments, the tangent may be written in the general form

$$\mathbf{S}_{ij} = c_1\,\mathbf{K}_{ij} + c_2\,\mathbf{C}_{ij} + c_3\,\mathbf{M}_{ij}$$

where the c_i are scalar quantities involving the integration parameters of the method selected and Δt. Thus, any one step integrator may be considered and will affect only the specification of the constants in the tangent. When *FEAP* solves a problem without transient loading (e.g., inertial loading as mass times acceleration) the velocities and accelerations are set to zero prior to calling the element subroutine. Consequently, in programming the steps to compute the residual `r` the inertia terms have no effect for static or quasi-static problems and may be included (generally there are very few additional operations involved to add these terms). The programming of the tangent array, however, must distinguish between cases in which transient (e.g., inertial) loads are present and those in which they are omitted. The different cases are implemented in *FEAP* by making appropriate assignments to the c_i parameters. To facilitate the programming of the tangent array returned in `s` for the various cases, a parameter array `ctan(3)` is passed to the subprogram in labeled common `eltran`. When the task parameter `isw` is 3, the values in the `ctan` array are interpreted according to Table 5.6.

Thus, in solid mechanics applications the tangent matrix is defined in an element routine as

$$\mathbf{S} = ctan(1)\,\mathbf{K} + ctan(2)\,\mathbf{C} + ctan(3)\,\mathbf{M}$$

where $\mathbf{K}$ is the stiffness matrix, $\mathbf{C}$ is the damping matrix, and $\mathbf{M}$ is the mass matrix. For non-linear applications these matrices normally are computed with respect to the current values of the available solution parameters. The values provided in the `ctan` array are set by *FEAP* according to the active transient solution option. For a static

Parameter	Description
ctan(1)	c_1: Multiplier of `s` matrix for `ul(i,j,1)` terms (e.g., stiffness matrix multiplier)
ctan(2)	c_2: Multiplier of `s` matrix for `ul(i,j,4)` terms (e.g., damping matrix multiplier)
ctan(3)	c_3: Multiplier of `s` matrix for `ul(i,j,5)` terms (e.g., mass matrix multiplier)

Table 5.6: Tangent Parameters

option both `ctan(2)` and `ctan(3)` are zero. For options integrating first order differential equations in time only `ctan(3)` will be zero. For options integrating second order differential equations in time all the parameters are non-zero.

In Appendix A an example of the stiffness matrix and residual for a 2-node truss user element in *FEAP* is presented in detail.

5.5 Setting Options in Elements

FEAP requires setting of some parameters and provides also setting of additional options within element tasks.

5.5.1 Task 1 Options

Setting active nodal equations

Often it is necessary to use several element types to perform an analysis. For example it may be necessary to use both truss and frame (bending resistant) elements to perform an analysis. As developed in Appendix A, the truss element has one degree-of-freedom for each spatial dimension, whereas, the frame element must have additional unknowns to represent the bending behavior. For nodes connected only to truss elements it is not necessary to have the additional degrees-of-freedoms active and a user would be required to specify restraint conditions for these nodes and degrees-of-freedom. By inserting the following lines of code into the truss element routine for the `isw = 1` task *FEAP* will automatically eliminate the degrees-of-freedom with values greater than `ndm` (the spatial dimension of the mesh).

```
do i = ndm+1,ndf
```

```
  ix(i) = 0
end do ! i
```

This avoids the need to specify appropriate fixed boundary conditions for the unused values.

Instead, if one wishes to set the active degrees-of-freedom at each individual node of an element it is necessary to dimension the array as `ix(ndf,*)`. In this form the first column corresponds to the global pattern described above and columns 2 to `nen+1` are associated with the local element nodes `1` to `nen`. The element degrees-of-freedom are then assigned to each node individually by assigning a `1` for an active degree-of-freedom or `0` for an inactive one. Note when using this option: Do not make changes to the first column of the *ix* array.

Example: 3-node element with 3-dof/node

Consider a problem with three degrees-of-freedom and three nodes on each element. It is desired to have degrees-of-freedom 1 and 3 active on node 2 and degree-of-freedom 2 active on nodes 1 and 3. This is accomplished by setting the `ix` array values as:

```
ix(1,2) = 0    ! For node 1
ix(2,2) = 1
ix(3,2) = 0

ix(1,3) = 1    ! For node 2
ix(2,3) = 0
ix(3,3) = 1

ix(1,4) = 0    ! For node 3
ix(2,4) = 1
ix(3,4) = 0.
```

Note that for `isw = 1` the `ix` parameter is not used to pass the nodal connection array but is used to return the list of unused degrees-of-freedom.

Setting element plot sequence

Utility routines are also provided to provide the necessary list of nodes needed to properly draw the mesh for each element type during plot outputs. The names of the routines available are listed in Table 5.7. Generally, *FEAP* can figure out which routine to call if the parameter `pstyp` is set to the spatial dimension of the plot. Thus for line plots one includes the statement

Routine Name	Description
PLTLN2	2-node line element
PLTLN3	3-node line element
PLTRI3	3-node triangular element
PLTRI6	6-node triangular element
PLTR10	10-node triangular element
PLQUD4	4-node quadrilateral element
PLQUD8	8 or 9-node quadrilateral element
PLTQ16	16-node quadrilateral element
PLTET4	4-node tetrahedron element
PLTET10	4-node tetrahedron element
PLBRK8	8-node brick element
PLBK27	27-node brick element
PLBKPQR	64-node brick element

Table 5.7: Element Plot Definition Subprograms

```
pstyp = 1   ! 1-d line plots
```

selection of the correct plot is then determined by the number of nodes on the elements. Similarly for surface plots one includes

```
pstyp = 2   ! 2-d surface plots
```

and for solid elements the statement

```
pstyp = 3   ! 3-d solid plots
```

If no plotting is wanted for the element the parameter is set as

```
pstyp = 0   ! No plots
```

Setting plot captions

The plot captions for contour plots may be set by the user adding the include file

```
include  'elcapt.h'  ! ecapt(50), dcapt(50)
```

which contains two arrays: `ecapt(50)` and `dcapt(50)` which replace the default captions for element variables (`PLOT STREss`) and nodal variables (`PLOT CONT`), respectively. For example, the caption for nodal degree of freedom 3 may be reset in the element `isw=1` part using the statement:

```
dcapt(3) = '    PRESSURE    ' ! up to 17 characters
```

Similarly, the caption for stress variable 1 may be changed to a force type using the caption

```
ecapt(1) = ' Axial Force: N ' ! up to 17 characters
```

Alternatively, the `ecapt(*)` may be defined in the `isw.eq.8` part of the element.

Setting maximum number of element projections

By default the number of element items that may be projected to nodes is limited to eight (8). This may be increased by add the include file

```
include 'strnum.h'
```

and setting the variable `istv` to the number desired. This should be set as follows:

```
istv = max(istv,<number element projections>)
```

to avoid loss from other element projections.

N.B. Be sure to also set `iste` in the plot projection module (See Task 8 Options).

Setting number of element equations

In some problems individual elements have solution parameters that are not associated with a node. Also, in some cases the parameters are associated with a Lagrange multiplier constraint which implies the global equations have initial zero diagonals. To facilitate these constraints when using the standard profile solver the equations are placed after all the equations of all nodal parameters on each element.

During input of material parameters the number of element parameters associated with each material set may be assigned to the parameter `nlm` which may be included using the statement

```
        include 'hdata.h'   ! nlm,plm
```

If partitions are used during problem solution the appropriate partition for the multipliers may be assigned to the parameter `plm` which is in the same include file.

1. Hint: The value of `nlm` should also be saved in one of the material parameters of the `d(n)` array and retrieved for other `isw` values using:

   ```
        nlm = nint(d(n))
   ```

   where '`n`' is the location saved.

The solution for parameter '`i`' in each element is returned in the local array `ule(i,1)`. The value at the previous time step is in `ule(i,2)`, and the last solution increment in `ule(i,3)`. These are returned to the element using

```
        include 'lmdata.h'   ! ule(100,3)
```

This is identical to the way nodal variables are ordered. If the parameters have inertial effects a user needs to perform these and manage using additional history variables.

Setting number of global equations

An element module may also set the number of global equations (see Sect. 5.11) during input of the material parameters. this is accomplished by setting the desired value in the parameter `nge` and add the include file

```
        include 'hdata.h'      ! nge,pge
```

If partitions are used in the problem solution the one to be used may be set in the parameter `pge` which is in the same include file.

The values of global equations are passed back to an element during problem solution in the array `ulg(*)` which is accessed using the include

```
        include 'lmdata.h'     ! ulg(100)
```

Routine Name	Description
`CKTRIS`	2-d 3, 6, 7 or 10 node triangle
`CKISOP`	2-d 4, 8, 9 or 16 node quadrilateral
`CKTETS`	3-d 4 or 10 node tetrahedron
`CKPYR5`	3-d 5 node pyramid
`CKWED6`	3-d 6 node wedge
`CKBRK8`	3-d 8, 27 or 64 node hexahedron

Table 5.8: Element Checking Subprograms

5.5.2 Task 2 Options

Mesh checking is performed using the solution command

```
CHECk
```

and is used to ensure, where possible, that the element connection array `IX` is correctly numbered and that the element area or volume is positive. Table 5.8 lists the basic routines that are available for use in checking 2 or 3-d solid elements. In some instances these routines will make changes to the ordering of nodes in the `IX` array to give proper ordering. It is recommended that after correction a new input file be created using

```
OUTM
```

It may be necessary to edit this file to add any missing parts.

The routines for 2-d checking are accessed form an `ELMTnn` module using the call

```
call <cktris,ckisop>(ix,xl,shp2(1,1),ndm)
```

and for 3-d by

```
call <cktets,ckbrks>(n_el,ix,xl,ndm,nel, shp3)
call <ckpyr5,ckwed6>(n_el,ix,xl,ndm, shp3)
```

The parameters require using include files

```
include 'qudshp.h'  ! shp2, shp3
include 'eldata.h'  ! n_el, nel
```

while the remainder are arguments of the `ELMTnn` module.

5.5.3 Task 3 Options

The basic structure of the element module for transient calculations also permits the calculation of eigenpairs when shifts are needed. That is when the command

```
        TANG EIGE ,, s
```

is used the problem

$$[\mathbf{K} - s\,\mathbf{M}]\;\mathbf{\Phi} = \mathbf{M}\,\mathbf{\Phi}\,\mathbf{\Lambda}$$

is needed. This is accomplished by setting `ctan(3) = - s * ctan(1)` and `ctan(2) = 0` and forming the element tangent as

$$S(:,:) = K(:,:) * ctan(1) + M(:,:) * ctan(3)$$

which is the required form for the transient solution (without damping) also.

5.5.4 Task 6 Options

The `TPLOt` solution command includes an option to save specific element quantities (e.g., stress,strain, etc.). This option is implemented for user elements by including the common

```
        real (kind=8) :: tt
        common /elplot/  tt(1000)
```

which is best set using

```
        include 'elplot.h'
```

and then inserting the statement

```
        tt(i) = value
```

at an appropriate location in the `isw = 3` task.

For example if it were desired to save the force and strain in the truss element the statements

```
tt(1) = EA*eps   ! Element axial force
tt(2) = eps      ! Element axial strain
```

could be placed anywhere after the stress and strain are defined. These values would be output by using a solution command sequence such as

```
batch
  tplot
end
stress,nn,1  ! saves force  for element nn
stress,nn,2  ! saves strain for element nn
show         ! writes tplot items to output file
```

Task 8 Options

The computation of element variables projected to nodes is carried out under `isw = 8`. This is described in the next section for a simple example. It is important when completing the projection module to inform *FEAP* how many parameters are being projected. This is accomplished in conjunction to the setting of `istv` in the `isw = 1` part by including

```
include 'setnum.h'
```

and then assigning the parameter `iste` the number of projected items [see Fig. 5.6].

5.6 Projection of element variables to nodes

The `STREss NODE n` solution command and the `PLOT STREss n` command require a projection of element variables to nodes.

For the solid elements these commands consider the parameter "`n=1,2,...,6`" to be the stresses in the order

$$\sigma_{11},\ \sigma_{22},\ \sigma_{33},\ \sigma_{12},\ \sigma_{23},\ \sigma_{31},$$

and the strains in the same order to be in `n=7,8,...,12`. Alternatively, the strains may be obtained using the commands `STRAin NODE n` or `PLOT STRAin n` with `n=1,2,...,6`. The stresses are also used to compute principal values which are output with the `STREss NODE` command and may be plotted using the command `PLOT PSTRess n` where the values are σ_n for `n=1,2,3` and are the invariants I_1, J_2, J_3 for `n=5,6,7`. The plotted value for $n = 4$ depends on the spatial dimension of the problem.

For other elements the values projected differ and programmers should consult the user manual or the source code of each element.

A continuous stress field is assumed to obtain the nodal values. Accordingly, each component is expressed as

$$\sigma_i = N_\alpha \, \tilde{\sigma}_\alpha$$

where σ_i is any value which is to be projected to nodes (e.g., a stress or strain), N_α are shape functions for the element type considered, and σ_α nodal values of the projected quantity.

Solid and thermal elements use a local least squares method to project stress, strain, and temperatues as described in the paper by Govindjee *et al.*[13] or in the monograph chapter by Mitchell *et al.*[14]. In this approach one first does a full least squares projection on each element individually using

$$M^e_{\alpha\beta} \tilde{\sigma}^e_\beta = \int_\Omega N_\alpha \, N_\beta \, \mathrm{d}\Omega = \int_\Omega N_\alpha \, \hat{\sigma}_i \, \mathrm{d}\Omega$$

and then averages the element nodal values as

$$\tilde{\sigma}_\beta = \frac{1}{E} \sum_{e=1}^{E} \tilde{\sigma}^e_\beta$$

where E is the number of elements at node β. The averaging step is performed automatically by *FEAP*.

Other elements use a diagonal weight matrix to project the values. For simple elements the matrix is computed by a procedure identical to mass lumping. For example,

$$M_{\alpha\alpha} = \int_\Omega N_\alpha \, \mathrm{d}\Omega$$

defines a 'row sum' form of projection matrix[15, 16, 17, 18]. Using the above results in the set of equations and a least square fit with the finite element values $\hat{\sigma}_i$ gives the equation set

$$M_{\alpha\alpha} \, \tilde{\sigma}_\alpha = \int_\Omega N_\alpha \, \hat{\sigma}_i \, \mathrm{d}\Omega \ .$$

This defines nodal values for projected quantities. Since the coefficient matrix is diagonal the solution to the set of equations for each component is trivial. The diagonal equation solution may also be performed automatically by *FEAP*.

Each of the above may be performed using the `P` and `S` element array when `isw = 8`. In the local least squares approach the values are returned as

$$\texttt{P}(\beta) = 1 \quad \text{and} \quad \texttt{S}(\beta, i) = \tilde{\sigma}_\beta$$

In the row sum algorithm the values are returned as

$$\texttt{P}(\alpha) = M_{\alpha\alpha} \quad \text{and} \quad \texttt{S}(\alpha, i) = \int_\Omega N_\alpha \, \hat{\sigma}_i \, \mathrm{d}\Omega$$

For the stress projection, the array for the projected quantities is dimensioned `S(nen,*)` and not `S(nst,*)`.

To permit each element to project its own quantities it is necessary to add the projection operations for each element under `ISW = 8`.[4] These are performed locally for each element similar to all other operations. Figure 5.6 shows a simple row sum routine for two-dimensional elements with 4-stress components begin projected. When multiple element types are used in an analysis users must be careful to project like quantities to common values of the `S(nen,*)` array so as to get correct results.

5.7 Elements with History Variables

FEAP provides options for each element to manage variables which must be saved during the solution. These are history variables and are separated into three groups: (a) Variables associated with the last converged solution time t_n; (b) Variables associated with the current solution time t_{n+1}; and variables which are not associated to any particular time. All history variables are associated with the allocation name `H` which has a pointer value 49. Users are not permitted direct access to the data stored as `H` (of course, it is possible to access from `hr(np(49))` but this should not normally be attempted!). Before calling the element routine for each element, *FEAP* transfers the required history variable to a local storage for each type. Users may then access the history data for each element and if necessary update values and return them *FEAP*. Only for specific actions will the local history data be transferred back to the appropriate `H` locations. The element history data associated with t_n starts at the memory address of the pointer for `NH1` using the double precision dummy array `HR` in blank common; similarly data for t_{n+1} starts at the memory address of the pointer for `NH2`, and that not associated with a time at `NH3`. The three pointers are passed to each element routine in the labeled common

[4]An implementation of the Zienkiewicz-Zhu projection method is implemented using `ISW = 24`.

```
      subroutine slcn2d(sig,shp,xsj,sg,lint,nel,nes, p,s)
!-----[--.----+----.----+----.-----------------------------------]
!      Purpose: Project element variables to nodes

!      Inputs:
!        sig(nes,*) - Stresses at quadrature points
!        shp(nel,*) - Shape functions at quadrature points
!        xsj(*)     - Volume element at quadrature points
!        sg(3,*)    - Gauss points (1,2) and weights (3)
!        lint       - Number of quadrature points
!        nel        - Number nodes on element
!        nes        - Dimension of stress array

!      Outputs:
!        p(nen)   - Weights for 'lumped' projection
!        s(nen,*) - Integral of variables
!-----[--.----+----.----+----.-----------------------------------]
      implicit  none

      include  'cdata.h'  ! Contains 'nen'
      include  'strnum.h' ! Contains 'iste'

      integer       :: i,l,lint,nel,nes
      real (kind=8) :: xsj(*),sig(nes,*),shp(nel,*),sg(3,*)
      real (kind=8) :: p(*),s(nen,*)

      do l = 1,lint
        do i = 1,nel
          p(i)     = p(i)     + shp(i,l)*xsj(l)
          s(i,1:4) = s(i,1:4) + sig(1:4,l)*shp(i,l)*xsj(l)
        end do ! i
      end do ! i
      iste = 4  ! Returns number projections

      end subroutine slcn2d
```

Figure 5.6: Element variable projection routine by row sum

```
      integer        nh1,nh2,nh3
      common /hdata/ nh1,nh2,nh3
```

5.7.1 Assigning amount of storage for each element

The specification for the amount of history information to be associated with each material set is controlled in the `isw = 1` task of an element routine. For each material type specified within the element routine a value for the length of the `NH1` and the `NH3` data must be provided (the amount of `NH2` data will be the same as for `NH1`). This is accomplished by setting the variables `nh1` and `nh2` in common `hdata` (see above) to the required values. That is, the statements required are:

```
if(isw .eq. 1) then
  . . .
  nh1 = 6
  nh3 = 10
  . . .
```

reserves 6 words of `NH1` and `NH2` data and 10 words of `NH3` data for each element with the current material number. Care should be taken to minimize the number of history variables since, for very large problems, the memory requirements can become large, thus reducing the size of problem that *FEAP* can solve.

Assigning storage for a user material

The storage for history parameters at each solution point in an element (usually a quadrature point) is assigned to the parameters `n1` and/or `n3` in the `umatin` module. These are then used to compute the total values for `nh1` and `nh3` in each element.

5.7.2 Accessing history data for each element

As noted above the data for each element is contained in arrays whose first word is located at `hr(nh1)`, `hr(nh2)` (where `nh1` and `nh2` are pointers) for t_n, t_{n+1}, respectively; and at `hr(nh3)` for that not associated with time (note that there are values for each only if non-zero values are assigned to `nh1` and/or `nh3` during the `isw = 1` task. Any other allocated data follows immediately after each first word It is a users responsibility to manage what is retained in each variable type; however, the order of placing the t_n and t_{n+1} data into the `NH1` and `NH2` arrays should be identical. There are no provisions to store integer history variables separately from double precision quantities. It is necessary to cast the integer data as double precision and move to the history location. For example, using the statement

```
hr(nh3+5) = dble(ivarbl)
```

saves the value for the integer variable `ivarbl` in the sixth word of the `NH3` element history array. At a subsequent iteration for this element the value of the integer would be recovered as

```
        ivarbl = int(hr(nh3+5))
```

While this wastes storage for integer variables, experience indicates there is little need to save many integer quantities and, thus, it was not deemed necessary to provide for integer history variables separately.

Although users may define new values for any of the `hr(nh1)`, `hr(nh2)`, or `hr(nh3)` types, the new quantities will be returned to the `H` history for the element only for `isw` tasks where residuals are being formed for a solution step (i.e., solution command `FORM`, `TANG,,1`, or `UTAN,,1` and for history reinitialization during a time update (i.e., solution command, `TIME`). These access the task options `isw` equal to 3 or 6 and 14, respectively.

If a user adds a new option for which it is desired to save the history variables, it is necessary to set the variables `hflgu` and `h3flgu` to true as required, if no update is wanted the variables should be set to false. These parameters are located in

```
        logical         hflgu,h3flgu
        common /hdatam/ hflgu,h3flgu
```

5.8 Accessing global array values

Usually, *FEAP* passes all the information needed to compute element arrays and results, either as arguments to the `elmtnn` routine or as variables in common blocks using the `include` statements. However, there are instances when other values may be useful. For example it may be useful to know which degree-of-freedoms are restrained by boundary conditions or have active equations. The information can be obtained using the `ix(*)` array for node numbers (one of the arguments to the `elmtnn` module) and the information in Table 3.1 for the `ID` array which has pointer `np(31)`. Based on the information in this table the array is retrieved using the code fragment

```
        include  'cdata.h'   ! numnp
        include  'pointer.h' ! np(400) and up(200)
        include  'comblk.h'  ! mr(*) and hr(*)
             ...
        call sub_name(ix, mr(np(31)), ndf, numnp)
```

where any name may be substituted for `sub_name`. Then in the module `sub_name` one has

```
subroutine sub_name(ix, id, ndf, numnp)
implicit   none
include   'eldata.h' ! nel
integer    ndf, numnp
integer    ix(nel), id(ndf,numnp,2)
```

It is now possible to look at each node in the `ix` array to know if the node is active (a zero value in `ix(*,*,2)`) or fixed (non-zero). In addition one may know the equation number of the active degree-of-freedoms by checking values in the `id(*,*,1)` part of the array (active equations are positive entries).

5.9 Elements with Finite Rotation Parameters

When considering structural elements that undergo large displacements it is usually necessary to treat the rotation parameters for large angle changes. The nodal parameters for this case are a rotation vector $\boldsymbol{\theta}$ and the finite rotations are given as an orthogonal matrix $\boldsymbol{\Lambda}$.

$$\boldsymbol{\Lambda}_{n+1} = \exp[\hat{\boldsymbol{\theta}}]\,\boldsymbol{\Lambda}_n$$

in which $\hat{\boldsymbol{\theta}}$ denotes a skew matrix given as

$$\hat{\boldsymbol{\theta}} = \begin{bmatrix} 0 & \theta_3 & -\theta_2 \\ -\theta_3 & 0 & \theta_1 \\ \theta_2 & -\theta_1 & 0 \end{bmatrix}$$

The actual method used to update the rotations and their increments must be specified when writing the element module `ELMTnn` and is performed by a user subprogram named `UROTmm` where `mm` is a number between `01` and `10`. To specify which routine is to be used, it is necessary to include the statement

```
rotyp = mm
```

in the section of `ELMTnn` where `isw = 1`. This parameter is located in the common `erotas` which has the structure:

```
 real*8          xln
 real*8          rots       ,rvel       ,racc       ,thkl
 integer                                                  rotyp
 common /erotas/ xln(9,9,4),
&                rots(3,9,2),rvel(3,9,2),racc(3,9,2),thkl(9),rotyp
```

The other entries in the common are arrays that return values for each element to treat the rotation values and rates. We shall return to their description after describing the treatment of the global nodal data for rotations.

5.9.1 Nodal rotation treatment: `UROTmm` subprogram

The nodal rotation data is stored in the array `xlg` which is dimensioned as

```
xlg(9,6,numnp)
```

For node `ng`, the entries in `xlg` are stored as follows:

Component	I/O	Description
`XLG(1:9,1,ng)`	I	Rotation matrix $\mathbf{\Lambda}_n$ at time t_n (Alternatively, entries 1 to 4 may be used to store a quaternion).
`XLG(1:9,2,ng)`	O	Rotation matrix $\mathbf{\Lambda}_{n_a}$ at time t_{n_a}
`XLG(1:9,3,ng)`	O	Rotation matrix $\mathbf{\Lambda}_{n+1}$ at time t_{n+1}
`XLG(1:3,4,ng)`	O	Rotation increment angle $\Delta\boldsymbol{\theta}$
`XLG(4:6,4,ng)`	I	Rotation rate $\boldsymbol{\omega}_n$ at time t_n
`XLG(7:9,4,ng)`	I	Rotation acceleration $\boldsymbol{\alpha}_n$ at time t_n
`XLG(1:3,5,ng)`	O	Rotation angle $\boldsymbol{\theta}$
`XLG(4:6,5,ng)`	O	Rotation rate $\boldsymbol{\omega}_{n+a}$ at time t_{n+a}
`XLG(7:9,5,ng)`	O	Rotation acceleration $\boldsymbol{\alpha}_{n+a}$ at time t_{n+a}
`XLG(1:9,6,ng)`	-	Rotation matrix $\mathbf{\Lambda}_0$ at time t_0

While storage is provided for the 3×3 rotation matrices the representation may also be specified in terms of quaternions for which only 4 components are necessary. In this case the 9 entries may be divided into two 4 entry quantities if required. Indeed, the space may be used in anyway necessary provided, no conflict in the way each time value is associated to the data. Note that sufficient storage is available to define integration methods for which the rotation is defined at an intermediate time t_{n+a}.

For a typical node `n` in the mesh the location of the entries in the `xlg` array are obtained from

```
ng = mropt(n,2)
```

and the routine `UROTmm` is called as:

```
      call urotmm(xlg(1,1,ng),xlg(1,2,ng),xlg(1,3,ng),
    &            xlg(1,4,ng),xlg(1,5,ng),
    &            xlg(4,4,ng),xlg(4,5,ng),
    &            xlg(7,4,ng),xlg(7,5,ng),du,tsw)
```

where `du(1:3)` are the solution increments for rotation from the solver and `tsw` is the time update switch which is set according to

```
      tsw = 1: Initialize for new time step
      tsw = 2: Update within a time step
      tsw = 3: Back up to beginning of time step
```

The entry $u(1)$ is the location for the first entry in the rotation vector $\boldsymbol{\theta}$.

5.9.2 Local nodal rotation treatment

When each element that is associated with nodal rotation parameters is called the rotation data is transferred to local storage in a manner similar to treatment of translations. The local data is passed to each element using the common `erotas` defined above. The entries in the local arrays are extracted from the global array according to:

```
      xln(1:9,nl,1)  = xlg(1:9,1,ng)
      xln(1:9,nl,2)  = xlg(1:9,2,ng)
      xln(1:9,nl,3)  = xlg(1:9,3,ng)
      xln(1:9,nl,4)  = xlg(1:9,6,ng)
      rots(1:3,nl,1) = xlg(1:3,4,ng)
      rots(1:3,nl,2) = xlg(1:3,5,ng)
      rvel(1:3,nl,1) = xlg(4:6,4,ng)
      rvel(1:3,nl,2) = xlg(4:6,5,ng)
      racc(1:3,nl,1) = xlg(7:9,4,ng)
      racc(1:3,nl,2) = xlg(7:9,5,ng)
```

where `nl` is a local node number between 1 and 9 (the maximum provided in the current `erotas` and `ng` is the global node number associated with each local number.

Using the above data structure one can program the updates in any manner that does not conflict with the time treatment. The only interface to `FEAP` is through how the increment `du(4:6,n)` is defined.

Component	Description
EPL(1) - EPL(3)	Linear momenta
EPL(4) - EPL(6)	Angular momenta
EPL(7)	Kinetic energy
EPL(8)	Stored energy
EPL(9)	Work by external loads
EPL(10)	Total energy

Table 5.9: Momenta and Energy Assignments

5.10 Energy Computation

FEAP elements provide an option to accumulate the total momenta and energy during the solution process. The values are accumulated in the array `EPL(20)` when the switch parameter `isw` is 13 and written to a file named `Pxxxx.ene` (where xxxx is extracted from the problem input filename) whenever the solution command **TIME** is used. The array `EPL(2)` is in the common block named `ptdat6` which has the structure:

```
real*8          epl
integer                   iepl,       neplts
common /ptdat6/ epl(20)0,iepl(2,200),neplts
```

For problems in solid mechanics the linear momenta are stored as follows:

The linear momenta are computed as:

$$\mathbf{p} \; = \; \int_\Omega \rho \, \mathbf{v} \; d\Omega$$

the angular momenta as:

$$\boldsymbol{\pi} \; = \; \int_\Omega (\mathbf{I} \, \boldsymbol{\omega} \; + \; \mathbf{x} \; \times \; \mathbf{p}) \, d\Omega$$

the kinetic energy

$$K \; = \; \int_\Omega \rho \, \mathbf{v} \cdot \mathbf{v} \; d\Omega$$

the stored energy as

$$U \; = \; \int_\Omega W(\mathbf{C}) \; d\Omega$$

and the work by external loads as

$$V \; = \; \int_\Gamma (\mathbf{x} - \mathbf{X}) \cdot \mathbf{F}_{ext} \; d\Gamma \; .$$

Array	Description
`U(NDF,NUMNP,1)`	Displacement at t_{n+1}^k
`U(NDF,NUMNP,2)`	Incremental Displacement at $t_{n+1}^k - t_n$
`U(NDF,NUMNP,3)`	Incremental Displacement at $t_{n+1}^k - t_{n+1}^{k-1}$
`UD(NDF,NUMNP,1)`	Velocity at t_{n+1}^k
`UD(NDF,NUMNP,2)`	Acceleration at t_{n+1}^k
	Additional arrays depend on time integrator

Table 5.10: Displacement and rate arrays at current solution state.

The value of the displacement and velocity at the current time t_{n+1} are passed in `ul(i,j,1)` and `ul(i,j,4)`, respectively. Note that this is true no matter which time integration algorithm is specified.

The local values are assigned from the global arrays for displacement, which has the pointer location `hr(np(40))` and often dimensioned as `u(ndf,numnp,*)`, and rates, which has the pointer location `hr(np(42))` and often dimensioned as `ud(ndf,numnp,*)` [see Table 5.10].

5.11 Global constraints on elements

In some cases it is necessary to add constraints that affect more than a single element in the mesh. Some constraints are applied directly to the elements. The specification of the input data for global equations is described in the user manual for *FEAP* (see, Sect. 5.15). The value of the number of global equations is stored in the integer variable `geqnum` and the partition to which it applies in the integer variable `gpart` and added to the common blocks accessed using the statement

```
      include 'pglob1.h'
```

This data is used to construct the matrix structure but is not needed directly to develop the contributions to elements.

Given the set of constraint equations $C_I(\tilde{\mathbf{u}}_a) = 0$, where implicitly we assume that the displacements affect at least several elements, the introduction using a perturbed

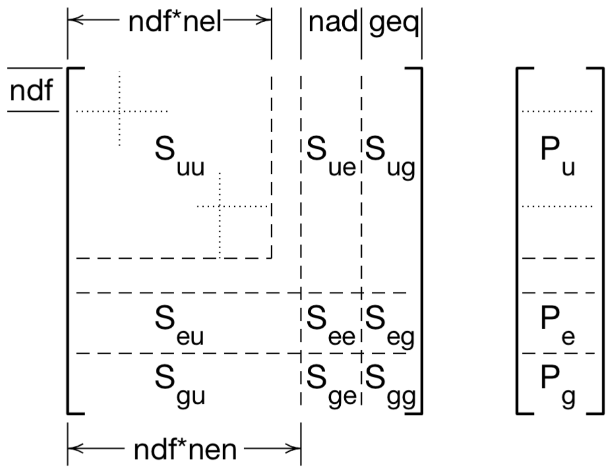


Figure 5.7: Tangent matrix and residual including element and global constraints.

Lagrangian approach may be written as[5]

$$\Pi_\lambda(\tilde{\mathbf{u}}_a, \lambda_I) = \lambda_I \left[C_I(\tilde{\mathbf{u}}_a) - \frac{1}{2k}\,\lambda_I \right]$$

The variation of the functional yields the result

$$\delta\Pi_\lambda = \delta\lambda_I \left[C_I(\tilde{\mathbf{u}}_a) - \frac{1}{k}\,\lambda_I \right] + \lambda_I \left[\frac{\partial C_I}{\partial \tilde{\mathbf{u}}_a}\,\delta\tilde{\mathbf{u}}_a \right]$$

The multiples of the variations are appended to each of the affected element residuals using

$$\begin{aligned} \delta\tilde{\mathbf{u}}_a^T \mathbf{P}_u^a &= -\delta\tilde{\mathbf{u}}_a^T \left(\frac{\partial C_I}{\partial \tilde{\mathbf{u}}_a} \right)^T \lambda_I \\ \delta\lambda_I P_\lambda^I &= -\delta\lambda_I \left[C_I(\tilde{\mathbf{u}}_a) - \frac{1}{k}\,\lambda_I \right] \end{aligned}$$

where the configuration for the terms is shown in Fig. 5.7. Note that the actual number of nodes on an element may be `nel` and be less than `nen`. Nevertheless, the global equations always occupy the locations shown based on `nen`. The values for the Lagrange multipliers is available in an element in the array `ule(100)` (current maximum for global constraints controlled by this include) which is included using

[5]The term with the penalty factor `k` may be omitted to give a classical Lagrange multiplier implementation.

Type	Logical	Values
Coordinate	`globxsc`	`gxscale`
Time	`globtsc`	`gtscale`
Mass	`globmsc`	`gmscale`
Displacement	`globdsc`	`gdscale(50)`
Element	`globesc`	`gescale(50)`

Table 5.11: Global scaling parameters in `pglob1.h`.

```
      include 'lmdata.h'
```

The remaining quantities (e.g., `nen`) are passed as arguments to the element or in include files as previously described.

The terms in the tangent matrix are deduced from

$$d(\delta\Pi_\lambda) = \delta\lambda_I \left[\frac{\partial C_I}{\partial \tilde{\mathbf{u}}_a} \, d\tilde{\mathbf{u}}_a - \frac{1}{k} \, \delta_{IJ} \, d\lambda_J \right] + \delta\tilde{\mathbf{u}}_a^T \left[\left(\frac{\partial C_I}{\partial \tilde{\mathbf{u}}_a} \right)^T d\lambda_I + \frac{\partial^2 C_I}{\partial \tilde{\mathbf{u}}_a \partial \tilde{\mathbf{u}}_b} \, d\tilde{\mathbf{u}}_b \, \lambda_I \right]$$

and give

$$\mathbf{S}_{ab} = \lambda_I \, \frac{\partial^2 C_I}{\partial \tilde{\mathbf{u}}_a \partial \tilde{\mathbf{u}}_b} \quad \mathbf{G}_{aJ} = \left(\frac{\partial C_I}{\partial \tilde{\mathbf{u}}_a} \right)^T$$

$$\mathbf{G}_{bI}^T = \left(\frac{\partial C_I}{\partial \tilde{\mathbf{u}}_b} \right) \qquad H_{IJ} = \frac{1}{k} \, \delta_{IJ}$$

5.12 Scaling factors for elements

When multi-physics problems are solved it may be necessary to scale equations into a non-dimensional form. *FEAP* permits the scaling factors to be specified using *global mesh commands* (See User Manual for details on specifying commands). The global scaling parameters are passed to routines in the program in the `pglob1.h` include file. The data consists of a logical flag and numeric values. If the data is input the logical flag is set to `.true.` otherwise it is false. The parameter names and flags are shown in Table 5.11.

5.13 Dynamic periodic response in elements

The solution of linear problems may be performed in frequency if the equations are written in complex arithmetic form (see *FEAP* User Manual section on periodic inputs on linear equations). Accordingly, we let the force be expressed as

$$\mathbf{F}(t) \;=\; \hat{\mathbf{F}}(\omega)\exp(i\,\omega\,t) \tag{5.1}$$

where $i = \sqrt{-1}$ and ω. The notation $\hat{(\cdot)}$ denotes a complex quantity. Thus, the intensity of the force is assumed to be a complex vector. Accordingly,

$$\begin{aligned} \mathbf{F}_r \;&=\; \Re(\hat{\mathbf{F}}) & (5.2)\\ \mathbf{F}_i \;&=\; \Im(\hat{\mathbf{F}})\;. & (5.3) \end{aligned}$$

For a linear problem the matrices $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ are constant and assuming a solution in the form:

$$\mathbf{u}(t) \;=\; \hat{\mathbf{u}}(\omega)\exp(i\,\omega\,t) \tag{5.4}$$

the equation of motion for a solid mechanics problem may be written as

$$\begin{aligned} \left[-\,\omega^2\,\mathbf{M} + i\,\omega\,\mathbf{C} + \hat{\mathbf{K}}\right]\;\hat{\mathbf{u}}(\omega) &= \hat{\mathbf{F}}(\omega)\\ \hat{\mathbf{A}}\;\hat{\mathbf{u}}(\omega) &= \hat{\mathbf{F}}(\omega) \end{aligned} \tag{5.5}$$

which may be solved for each specified frequency and load to give a solution for the $\hat{\mathbf{u}}(\omega)$.

To program the above in an element routine in *FEAP* the element array for the stiffness is dimensioned as `s(nst,nst,2)` and that for the residual as `p(nst,2)`. The real parts are stored in `s(nst,nst,1)` and `p(nst,1)` and the imaginary parts in `s(nst,nst,2)` and `p(nst,2)`.

5.13.1 Viscoelastic damping

For a linear viscoelastic material the stiffness matrix in the frequency domain is written in terms of *complex moduli*. Accordingly, for this case the element stiffness matrix is also complex and may be expressed as

$$\hat{\mathbf{K}} = \mathbf{K}_r + i\,\mathbf{K}_i \tag{5.6}$$

Thus, the element array is computed as

$$\begin{aligned} \mathbf{K}_r &= \int_{\Omega_e} \mathbf{B}^T\mathbf{D}_r\mathbf{B}\,\mathrm{d}\Omega\\ \mathbf{K}_i &= \int_{\Omega_e} \mathbf{B}^T\mathbf{D}_i\mathbf{B}\,\mathrm{d}\Omega \end{aligned} \tag{5.7}$$

The residual is obtained by computing the real and imaginary parts of the strain as

$$\begin{aligned}\boldsymbol{\epsilon}_r &= \mathbf{B}\,\mathbf{u}_r \\ \boldsymbol{\epsilon}_i &= \mathbf{B}\,\mathbf{u}_i\end{aligned} \tag{5.8}$$

and premultiplying by the complex moduli as

$$\hat{\mathbf{p}} = \int_{\Omega_e} \mathbf{B}^T \left[\mathbf{D}_r + i\,\mathbf{D}_i\right] \left(\boldsymbol{\epsilon}_r + i\,\boldsymbol{\epsilon}_i\right)\,\mathrm{d}\Omega \tag{5.9}$$

to obtain

$$\begin{aligned}\mathbf{p}_r &= \int_{\Omega_e} \mathbf{B}^T \left[\mathbf{D}_r\boldsymbol{\epsilon}_r - \mathbf{D}_i\boldsymbol{\epsilon}_i\right]\,\mathrm{d}\Omega \\ \mathbf{p}_i &= \int_{\Omega_e} \mathbf{B}^T \left[\mathbf{D}_r\boldsymbol{\epsilon}_i - \mathbf{D}_i\boldsymbol{\epsilon}_r\right]\,\mathrm{d}\Omega\end{aligned} \tag{5.10}$$

Alternatively, the calculations may be performed from the element stiffness matrix parts as

$$\begin{aligned}\mathbf{p}_r &= \mathbf{K}_r\mathbf{u}_r - \mathbf{K}_i\mathbf{u}_i \\ \mathbf{p}_i &= \mathbf{K}_r\mathbf{u}_i + \mathbf{K}_i\mathbf{u}_r\end{aligned} \tag{5.11}$$

In either case, all the *FEAP* elements for solids and structures compute the first term as part of the standard residual. Similarly, all the elements compute the real part of the stiffness matrix.

5.13.2 Rayleigh damping

In Rayleigh damping we express the damping matrix as

$$\hat{\mathbf{C}} = a_0\mathbf{M} + a_1\hat{\mathbf{K}} \tag{5.12}$$

thus, the complex coefficient matrix becomes

$$\hat{\mathbf{A}} = -\omega^2\mathbf{M} + i\,\omega\,\left[a_0\mathbf{M} + a_1\left(\mathbf{K}_r + i\,\mathbf{K}_i\right)\right] + \mathbf{K}_r + i\mathbf{K}_i \tag{5.13}$$

giving the real and imaginary parts as

$$\begin{aligned}\mathbf{A}_r &= -\omega^2\mathbf{M} - \omega\,a_1\,\mathbf{K}_i + \mathbf{K}_r \\ \mathbf{A}_i &= \omega\left(a_0\mathbf{M} + a_1\mathbf{K}_r\right) + \mathbf{K}_i\end{aligned} \tag{5.14}$$

5.14 Using `formfe` to add element functions

Access to all element operations is carried out by a call to the module `formfe`. Users may add new element functions using this access as:

```
call formfe(np(40),pnd,pnd,pnd,dfl,dfl,dfl,dfl,isw,nl1,nl2,nl3)
```

where the arguments are defined in Table 5.12.

NAME	Description
`np(40)`	Pointer in `hr(*)` array for current solution values
`pnd,`	Dummy pointers (can be `np(26)`).
`dfl`	Logical flags (set to `.false`)
`isw`	Element operation parameter (should be > 30
`nl1`	First element to process
`nl2`	Last element to process
`nl3`	Increment in element counter (usually 1

Table 5.12: Argument parameters for calls to `formfe`.

Chapter 6

UTILITY ROUTINES

The *FEAP* system includes several subprograms that can assist developers in writing new modules. In the next sections we describe some of the routines which perform numerical integration, compute shape functions and their derivatives, etc.

6.1 Numerical quadrature routines

Details on quadrature formula types and the layout and location of points and weights may be found in standard references.[19, 20, 21, 15, 16, 17, 18] Here only the description of subroutine calls is included together with the available options on number of points.

6.1.1 One dimensional quadrature

Line integrals may be evaluated using Gaussian quadrature in which the approximation to an integral is given as

$$\int_{-1}^{+1} f(\xi)\,\mathrm{d}\xi \approx \sum_{l=1}^{L} f(\xi_l)\,W_l \tag{6.1}$$

where ξ_l are quadrature *points* and W_l are the *weights* to be applied at each point. The weights satisfy the condition.

$$\sum_{l=1}^{L} W_l = 2\;. \tag{6.2}$$

The Gauss-Legendre formula has points $|\xi_l|$ which are all less than unity. The subprogram call

```
        CALL INT1D ( L , SW )
```

in which `L` is assigned an integer value between 1 and 5 returns the points and weights are returned in the two dimensional array `SW(2,*)` of type `REAL*8`: Points in `SW(1,*)` and weights in `SW(2,*)`. The Gauss-Legendre formula integrates exactly polynomials up to order `2*L - 1`.

The Gauss-Lobato formula has two of its points at `-1` and `1` with the remainder in the interior of the interval. A routine to perform quadrature is obtain by using the call

```
        CALL INT1DL ( L , SW )
```

in which `L` is assigned an integer value between 1 and 6. The values of the points and weights are returned in the two dimensional array `SW`: Points in `SW(1,*)` and weights in `SW(2,*)`.

6.1.2 Two dimensional quadrature

Two dimensional quadrature on quadrilateral domains may be performed by repeated one-dimensional integration. The two dimensional integrations are approximated by

$$\iint_{-1}^{+1} f(\xi,\eta)\,\mathrm{d}\xi\,\mathrm{d}\eta \approx \sum_{l=1}^{L} f(\xi_l,\eta_l)\,W_l \qquad (6.3)$$

where L is the total of all quadrature points. A routine to compute $n \times n$ order Gauss-Legendre quadrature is obtained by the call

```
        CALL INT2D ( L , LINT, SW )
```

where `L` is assigned to the number of points in *each direction*, `LINT` is returned as the total number of points and `SW(3,*)` is an array containing the points and weights according to: `SW(1,l)` contains values of the points ξ_l; `SW(2,l)` contains values of the points η_l; and `SW(3,l)` contains values of the weights W_l.

Two dimensional quadrature on triangles may be performed using the subprograms call

Type	Number Points	Location
1	1	Centroid $(O(h^2))$
3	3	Mid-sides $(O(h^3))$
-3	3	Interior $(O(h^3))$
4	4	Interior $(O(h^4))$ - Negative Wt.
6	6	Nodal $(O(h^3))$
-6	6	Interior $(O(h^4))$
7	7	Interior $(O(h^6))$
-7	7	Nodal $(O(h^4))$
12	12	Interior $(O(h^7))$
13	13	Interior $(O(h^8))$ - Negative Wt.

Table 6.1: Quadrature for triangles

```
        CALL TINT2D ( L , LINT, SW )
```

where L is a type indicator, `LINT` returns the number of points, and `SW(4,*)` is an array which returns three area coordinates and the quadrature weight: `SW(1,l)` returns the area coordinate L_{1l} (as defined in [15, 16, 17, 18]); `SW(2,l)` returns the area coordinate L_{2l}; `SW(3,l)` returns the area coordinate L_{3l}; `SW(4,l)` returns the weight W_l; Table 6.1 describes the admissible types, number and location of quadrature points.

6.1.3 Three dimensional quadrature

Three dimensional quadrature on brick domains may be performed by repeated one-dimensional integration. The three dimensional integrations are approximated by

$$\iiint_{-1}^{+1} f(\xi,\eta,\zeta)\,\mathrm{d}\xi\,\mathrm{d}\eta\,\mathrm{d}\zeta \approx \sum_{l=1}^{L} f(\xi_l,\eta_l,\zeta)\,W_l \tag{6.4}$$

where L is the total of all quadrature points. A routine to compute $n \times n \times n$ order Gauss-Legendre quadrature is obtained by the call

```
        CALL INT3D ( L , LINT, SW )
```

where `L` is assigned to the number of points in *each direction*, `LINT` is returned as the total number of points and `SW(4,*)` is an array containing the points and weights according to: `SW(1,l)` contains values of the points ξ_l; `SW(2,l)` contains values of the

Type	Number Points	Location
1	1	Centroid $(O(h^2))$
-1	4	Nodal $()(h^2))$
2	4	Interior $(O(h^3))$
3	5	Interior $(O(h^4))$ - Negative wt.
4	11	Interior $(O(h^4))$ - Negative wt.
-4	11	Nodal $(O(h^3))$
5	14	Interior $(O(h^5))$
6	16	Interior $(O(h^5))$
8	8	Node & Face $(O(h^2))$

Table 6.2: Quadrature for tetrahedra

points η_l; and `SW(3,l)` contains values of the points ζ_l; and `SW(4,l)` contains values of the weights W_l.

Three dimensional quadrature on tetrahedra may be performed using the subprograms call

```
      CALL TINT3D ( L , LINT, SW )
```

where L is a type indicator, `LINT` returns the number of points, and `SW(5,*)` is an array which returns three area coordinates and the quadrature weight: `SW(1,l)` returns the volume coordinate $L_{1,l}$ (as defined in [15, 16, 17, 18]); `SW(2,l)` returns the volume coordinate $L_{2,l}$; `SW(3,l)` returns the volume coordinate $L_{3,l}$; `SW(4,l)` returns the volume coordinate $L_{4,l}$; `SW(5,l)` returns the weight W_l; Table 6.2 describes the admissible types, number and location of quadrature points.

6.2 Shape function subprograms

Finite element approximations commonly use shape function subprograms to perform computations of the functions and their derivatives at preselected points (often the quadrature points). *FEAP* includes options to obtain the shape functions for some low order elements (linear and quadratic order) in one and two dimensions and linear shape functions for three dimensions. In addition a cubic Hermitian interpolation routine is available. The calling arguments for routines is summarized below.

6.2.1 Shape functions in one-dimension

The shape functions for one dimensional elements, as shown in Fig. 6.1, may be computed using the shape function routines described below.

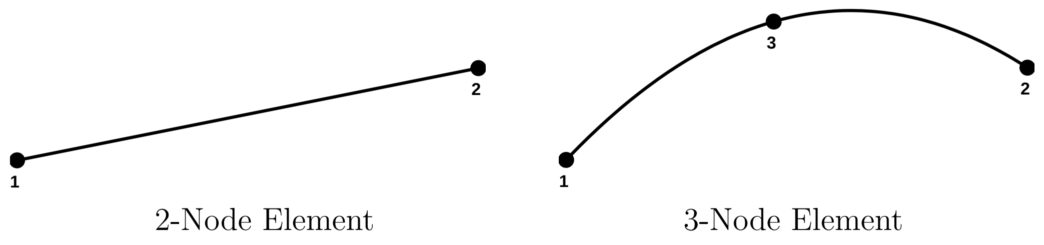


Figure 6.1: Line type elements in *FEAP* library

Lagrangian interpolation in one-dimensional isoparametric forms may be obtained using the call

```
        CALL SHP1D ( S , XL , SHP, NDM, NEL, XJAC )
```

where

Parameter	Description
`S`	Natural coordinate ξ
`XL(NDM,*)`	Nodal coordinates for element
`NDM`	Spatial dimension of mesh
`NEL`	Number element nodes (2 or 3)
`SHP(2,NEL)`	Shape function and derivative
`XJAC`	Jacobian transformation

The shape functions are evaluated as: `SHP(1,i)` shape function derivative along the axis of the element and `SHP(2,i)` the shape function N_i. In calculations integrals are represented as

$$\int_L f(N_i, N_{i,s})\,\mathrm{d}s = \int_{-1}^{1} f[N_i(\xi), N_{i,s}(\xi)]\,XJAC(\xi)\,\mathrm{d}\xi \tag{6.5}$$

and quadrature may be used for evaluation.

Calculation of natural coordinate derivatives only may be obtained with the call

```
        CALL SHAP1DN( S , SHP, NEL )
```

where

Parameter	Description
`S`	Natural coordinate ξ
`SHP(2,NEL)`	Shape function and derivative
`NEL`	Number element nodes (2 or 3)

where `SHP(1,i` contains $N_{i,\xi}$ and `SHP(2,i)` the shape function N_i.

Cubic Hermitian interpolation (e.g., for use in straight linear beam elements) given by

$$w = N_1^w \, \bar{w}_1 + N_2^w \, \bar{w}_2 + N_1^\theta \, \bar{\theta}_1 + N_2^\theta \, \bar{\theta}_2 \tag{6.6}$$

is obtained using the call

```
        CALL SHP1DH ( S , LEN , SHPW, SHPT )
```

where

Parameter	Description
`S`	Natural coordinate ξ
`LEN`	Length of the element (2-node)
`SHPW(4,2)`	Shape functions for w_i
`SHPT(4,2)`	Shape functions for θ_i

The arrays are evaluated as follows:

1. `SHPW(1,i), SHPT(1,i)` are first derivatives (e.g. $N_{i,x}$);
2. `SHPW(2,i), SHPT(2,i)` are second derivatives (e.g. $N_{i,xx}$);
3. `SHPW(3,i), SHPT(3,i)` are third derivatives (e.g. $N_{i,xxx}$); and
4. `SHPW(4,i), SHPT(4,i)` are shape functions (e.g. N_i).

6.2.2 Shape functions in two-dimensions

The shape functions for two dimensional triangular elements, as shown in Fig. 6.2, and quadrilateral elements, as shown in Fig. 6.3, may be computed using the shape function routines described below.

Two-dimensional C_0 isoparametric interpolation on quadrilaterals of linear, quadratic and cubic order may be obtained using the subprogram call

```
        CALL SHP2D ( SS, XL, SHP, XJAC, NDM, NEL, IX, FLG )
```

where

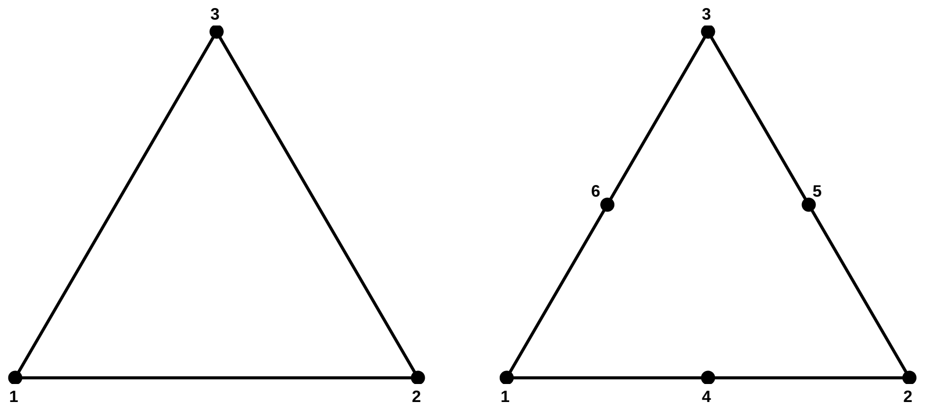
Parameter	Description
`SS(2)`	Natural coordinates ξ, η
`XL(NDM,NEL)`	Element coordinates in local order
`NDM`	Spatial dimension mesh (2 or 3)
`NEL`	Number nodes on element (4-9, 12, 16)
`IX(NEL)`	Element global node numbers
`FLG`	Return $\xi - \eta$ derivatives if true or
	$x - y$ derivatives if false
`SHP(3,NEL)`	Shape functions and derivatives
`XJAC`	Jacobian transformation from $x - y$ to $\xi - \eta$.

The array `SHP` stores the values in the order: `SHP(1,i)` derivative with respect to ξ or x; `SHP(2,i)` derivative with respect to η or y; `SHP(3,i)` shape function.

Two-dimensional C_0 isoparametric interpolation on triangles of linear, quadratic and cubic order may be obtained using the subprogram call

```
        CALL TRISHP ( SS, XL, NDM, IORD, XJAC, SHP )
```

where



3-Node Simplex 6-Node Element

Figure 6.2: Triangular surface type elements in *FEAP* library

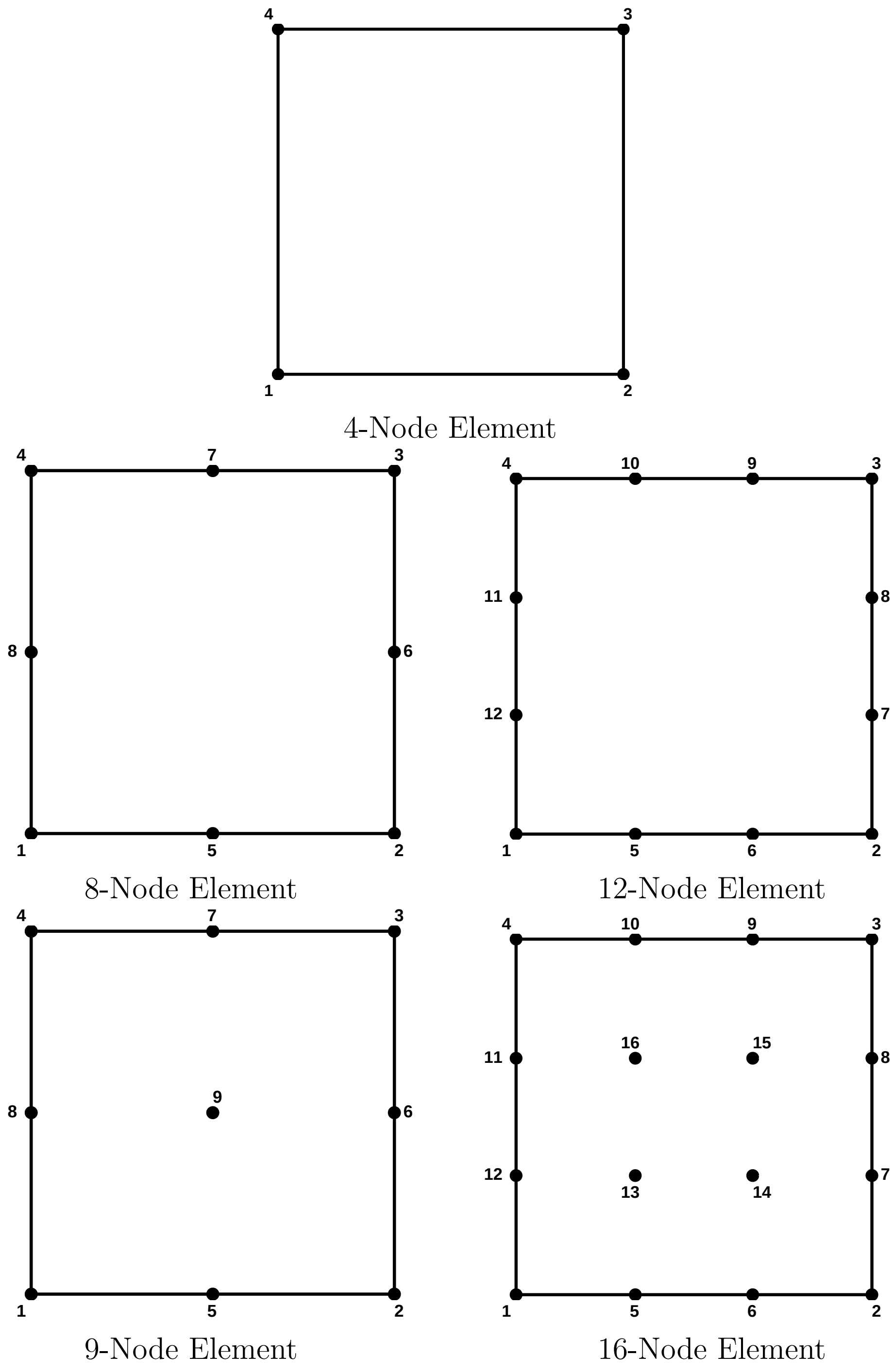


Figure 6.3: Quadrilateral surface type elements in *FEAP* library

Parameter	Description
`SS(3)`	Area coordinates L_1, L_2, L_3
`XL(NDM,*)`	Element coordinates in local order
`NDM`	Spatial dimension mesh (2 or 3)
`IORD`	Order of interpolation (1= 3-node,2 = 6-node, 3 = 7-node, 4 = 6-node + 3 bubble, 10 = 10-node cubic)
`XJAC`	Jacobian transformation from $x - y$ to $\xi - \eta$
`SHP(3,NEL`	Shape functions and derivatives

The array SHP stores the values in the order: SHP(1,i) derivative with respect to ξ or x; SHP(2,i) derivative with respect to η or y; SHP(3,i) shape function. The parameter IORD defines the order of interpolation. If it is 1 simple 3-node triangles with linear interpolation is returned; if 2 quadratic interpolation; if 3 the interpolation is generated plus a cubic bubble in the seventh function. Giving the IORD parameter as a negative returns hierarchical form for mid side nodes.

6.2.3 Shape functions in three-dimensions

The shape functions for three dimensional tetrahedral elements, as shown in Fig. 6.4, and brick elements, as shown in Fig. 6.5, may be computed using the shape function routines described below.

Three-dimensional C_0 isoparametric interpolation on bricks of linear order (i.e., 8-node elements) may be obtained using the subprogram call

```
        CALL SHP3D ( SS, XJAC, SHP, XL, NDM, NEL )
```

where

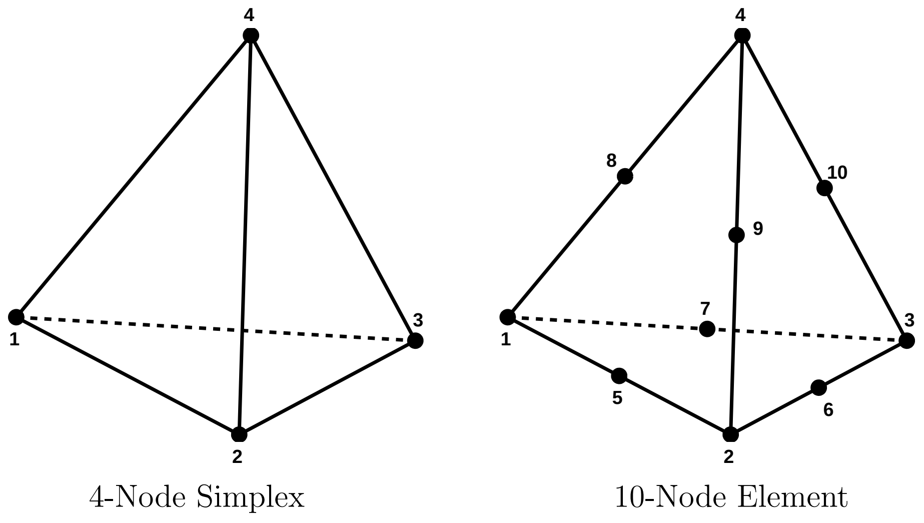
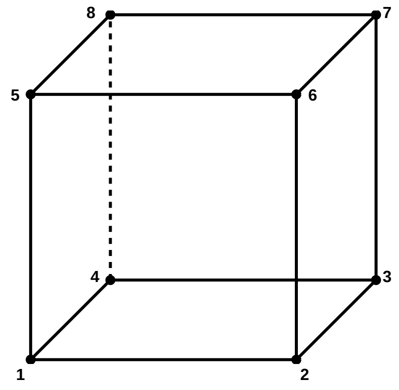
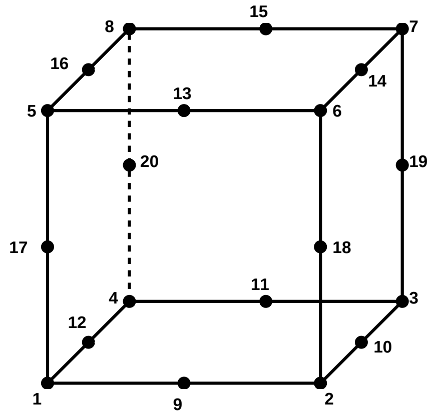
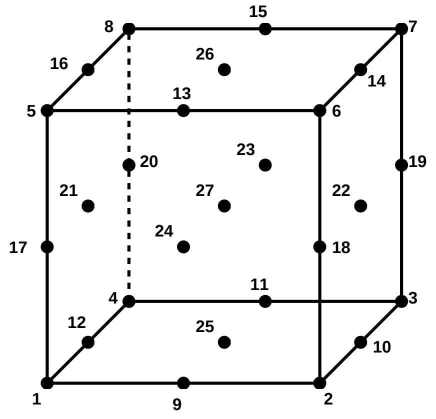


Figure 6.4: Tetrahedron solid type elements in *FEAP* library



8-Node Element





20-Node Element

27-Node Element

Figure 6.5: Brick solid type elements in *FEAP* library

Parameter	Description
`SS(3)`	Natural coordinates ξ, η, ζ
`XL(NDM,8)`	Element coordinates in local order
`NDM`	Spatial dimension mesh (2 or 3)
`NEL`	Number nodes on element: 8 = linear brick; 20 = serendipity quadratic; 27 = Lagrangian quadratic; 64 = Lagrangian cubic
`SHP(4,8)`	Shape functions and derivatives
`XJAC`	Jacobian transformation from xyz to $\xi\eta\zeta$.

The array `SHP` stores the values in the order: `SHP(1,i)` derivative with respect to x; `SHP(2,i)` derivative with respect to y; `SHP(3,i)` derivative with respect to z; `SHP(4,i)` shape function.

Three-dimensional C_0 isoparametric interpolation on tetrahedra of linear order (i.e., 4-node elements) may be obtained using the subprogram call

```
        CALL TETSHP ( SS, XL, NDM, NEL, XJAC, SHP )
```

where

Parameter	Description
`SS(4)`	Volume coordinates L_1, L_2, L_3, L_4
`XL(NDM,4)`	Element coordinates in local order
`NDM`	Spatial dimension mesh (3)
`NEL`	Number of nodes on element (4, 10, 11, 14, 15)
`XJAC`	Jacobian transformation from xyz to $\xi\eta\zeta$
`SHP(4,4`	Shape functions and derivatives

The array `SHP` stores the values in the same order as for the brick element.

6.3 Eigenvalues for 3×3 matrix

Three dimensional problems often require the solution of a 3×3 eigenproblem to generate principal values and directions. *FEAP* includes a special routine to calculate the values and vectors for symmetric arrays. The routine is used by a call to the subprogram as

```
        CALL EIG3 ( V, D, ROT )
```

On call to the routine `V(3,3)` is a `REAL*8` array containing the symmetric array to be diagonalized. On return the eigenvalues are contained in `D(3)` and the vectors for each value in the columns of the `V` array. A Jacobi method is used with `ROT` an integer parameter returning the number of rotations to diagonalize. The routine is quite efficient compared to any attempt to compute vectors after closed form solution of the cubic for roots.

In addition to the general eigensolution above *FEAP* includes options to compute principal values of a symmetric second order tensor for two and three dimensional problems. In two dimensional use, the call to

```
        CALL PSTR2D ( SIG, PV )
```

is used where `SIG(4)` stores stresses in the order σ_{11}, σ_{22}, σ_{33}, σ_{12} and returns principal values and directions in PV(3) in the order σ_1, σ_2, and θ, where the angle is in degrees between x and the 1-axis. This routine does not use `SIG(3)`.

In three dimensions the principal values are obtained using the call

```
        CALL PSTR3D ( SIG, PV )
```

where `SIG(6)` stores stresses in the order σ_{11}, σ_{22}, σ_{33}, σ_{12}, σ_{23}, σ_{31}, and returns principal values in PV(3) in the order σ_1, σ_2, σ_3. Roots are ordered from most positive to most negative.

6.4 Plot routines

Several options exist in the *FEAP* system to create graphical plots for data and results.

6.4.1 Mesh plots

FEAP has plot capabilities to represent some standard element shapes (provided element numbering is according to the standard *FEAP* convention - see User Manual for numbering). By default user elements are set to produce *no plot of any mesh data*. To add a capability for plotting standard elements it is necessary to set the parameter `pstyp` within the `ISW = 1` part of the element routine. To access the parameter `pstyp` it is necessary to include the common statement using

```
        include 'eldata.h'
```

For continuum elements where the shape of the element is identical to the space dimension of the mesh the parameter may be set as

```
        pstyp = ndm
```

However, if the dimension of the element topology is different from the mesh dimension it is necessary to explicitly state the dimension. For example, in a three dimensional problem where `NDM = 3` and the element topology is two dimensional the statement is given as

```
        pstyp = 2
```

Provided the nodal numbering of an element is as described in the *FEAP* User manual (i.e., numbered with vertex nodes first, followed by mid-side nodes, then face nodes and finally internal nodes) the program can use the actual number of nodes on the element to draw each element.

Failure to include a `pstyp` statement may result in unpredictable plots of the mesh and contour values.

The known types of plots for `pstyp = 1` are

1. Point element with one node obtained by call

   ```
   CALL PLTPT1 ( IEL )
   ```

2. Line element with two nodes obtained by call

   ```
   CALL PLTLN2 ( IEL )
   ```

   and for three node elements

   ```
   CALL PLTLN3 ( IEL )
   ```

The known types of plots for `pstyp = 2` are:

1. Triangular element with 3-nodes obtained by call

   ```
   CALL PLTRI3 ( IEL )
   ```

   and for 6-nodes obtained by call

   ```
   CALL PLTRI6 ( IEL )
   ```

2. Quadrilateral element with 4-nodes obtained by call

   ```
   CALL PLQUD4 ( IEL )
   ```

   for 8- or 9-node elements the plot call is

   ```
   CALL PLQUD8 ( IEL )
   ```

   and for 12- or 16-node quadrilaterals the call is

```
            CALL PLTQ16 ( IEL )
```

The known types of plots for `pstyp = 3` are:

1. Tetrahedral element with 4-nodes obtained by call

   ```
         CALL PLTET4 ( IEL )
   ```

   and for 10-node tetrahedra the call is

   ```
         CALL PLTET10( IEL )
   ```

2. Brick element with 8-nodes obtained by call

   ```
         CALL PLBRK8 ( IEL )
   ```

   and for 20- or 27-node bricks the call is

   ```
         CALL PLBRK27( IEL )
   ```

Using the above and internal extraction of element surfaces the program is able to make some hidden surface plots in three dimensions.

6.4.2 Element data plots

Users may construct plots within their elements (i.e., an `ELMTnn`) and access using the plot command:

```
       PLOT,PELE,v1,v2,v3
```

In interactive mode in the plot environment it is only necessary to enter

```
       PELE,v1,v2,v3
```

The values entered in `v1,v2,v3` are optional and are passed to the element through a common block as

```
       REAL*8          ELPLT
       COMMON /ELPDAT/ ELPLT(3)
```

The PELE option calls each element with the switch parameter `ISW = 20`. Users merely code whatever option they wish to include within their element module.

The standard color table is available through use of the subroutine call

```
        CALL PPPCOL(ICOL, 0)
```

in which ICOL designates the color to be assigned according to Table 6.3. An exception occurs for PostScript outputs where black and white are switched (since the background then is assumed to be white).

ICOL	COLOR	ICOL	COLOR
0	Black	10	Green-yellow
1	White	11	Wheat
2	Red	12	Royal blue
3	Green	13	Purple
4	Blue	14	Aquamarine
5	Yellow	15	Violet-red
6	Cyan	16	Dark slate blue
7	Magenta	17	Grey
8	Orange	18	Light grey
9	Coral		

Table 6.3: Color pallet for *FEAP* plots

A straight line segment may be drawn to the screen in the current color between the coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2) using the commands

```
        CALL PLOTL(X1,Y1,Z1, 3)
        CALL PLOTL(X2,Y2,Z2, 2)
```

Here the basic command is

```
        CALL PLOTL(Xi,Yi,Zi, IP)
```

where the three Cartesian coordinates relate to mesh coordinates (not screen values) and IP is a parameter defined according to Table 6.4.

The perimeter of a panel is drawn with standard line drawing commands starting with

IP	Action
1	Start panel fill
2	Move to point
3	Draw to point

Table 6.4: Values for control of plots

```
CALL PLOTL(X1,Y1,Z1, 1)
```

and continuing with a sequence of draw commands

```
CALL PLOTL(Xi,Yi,Zi, 2)
```

(however, no lines appear on the screen) and the fill of each panel is completed by the statement

```
CALL CLPAN
```

It should be noted that all plots within *FEAP* are performed in three dimensions. For two dimensional problems no z_i coordinates are available in the `XL(NDM,NEN)` array and, hence, it is necessary to assign zero values for the z_i coordinates before calling a plot subprogram. If a perspective view has been requested a full use of a x_i, y_i, z_i specification is made. In this case a user may wish to pass the value of some solution variable as the z_i value (scaled so that it will make sense relative to the x_i, y_i coordinate values). Similarly, if deformed plots are being performed it is necessary to add (scaled) displacements to the coordinates. The current value of the scaling parameter (i.e., variable `CS`) is available in labeled common `PVIEW`. In this case one can add the statements (assuming here that the displacements correspond to the coordinate directions)

```
DO NE = 1,NEL
  DO I = 1,NDM
    XP(I,NE) = XL(I,NE) + CS*UL(I,NE)
  END DO ! I
END DO ! NE
```

(`NEL` is the number of connected nodes to each element and is passed through labeled common `ELDATA`) before performing any deformed plots and then plot the appropriate values of `XP`. Indeed, this may always be performed as the value of `CS` will be zero for an *undeformed* plot.

6.4.3 Other user plots

It is also possible for users to prepare plot outputs unrelated to elements. The plot command

```
PLOT UPLOt v1 v2 v3
```

initiates a call to the subroutine `UPLOT` which has the basic structure

```
SUBROUTINE UPLOT(CT)
IMPLICIT   NONE
REAL*8     CT(3)
  ...
END
```

The argument `CT` contains the values for the three parameters `v1, v2, v3`. The default color is *white*. Direct plots in screen coordinates [lower left at (0,0); upper right at (1,1)] may be given using the statement

```
CALL DPLOT(XS,YS, IP)
```

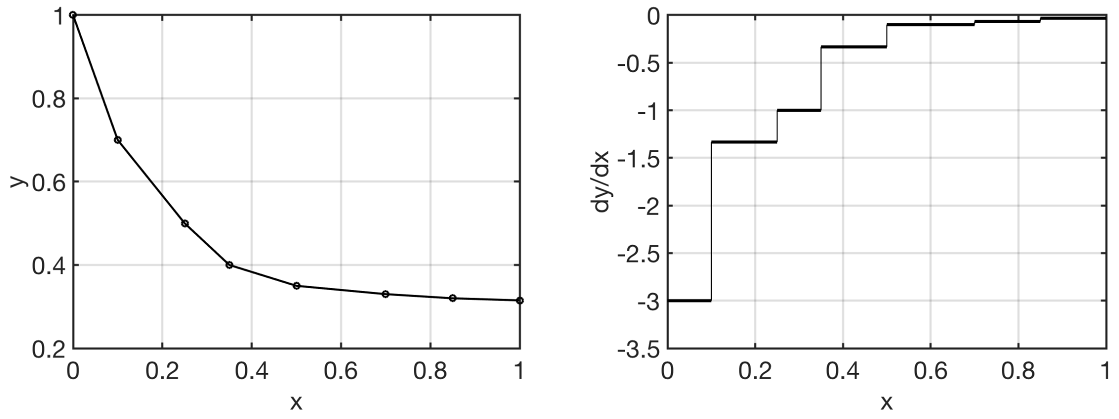
where `XS, YS` are between zero (0) and one (1) and `IP` is interpreted according to Table 6.4. Panels are closed using

```
CALL CLPAN
```

and colors treated according to values specified in calls to `PPPCOL`.

6.5 Tabular data

In some instances the parameters for loads or material data may be in tabular form. For example, a set of x-y data is shown in Fig. 6.6(a) as a set of piecewise linear data between the data points. The derivatives are constant between the data points as shown in Fig. 6.6(b).



(a) x-y data (b) dy/dx data

Figure 6.6: Example of tabular x-y data

If the data is stored in an array `xy_val(2,8)` as

$$\text{xy\_val(i,j)} = \begin{bmatrix} 0.00 & 1.000 \\ 0.10 & 0.700 \\ 0.25 & 0.500 \\ 0.35 & 0.400 \\ 0.50 & 0.350 \\ 0.70 & 0.330 \\ 0.85 & 0.320 \\ 1.00 & 0.315 \end{bmatrix} \tag{6.7}$$

the table may be initialized with a single call to the module

```
call dy_dx_table(xy_val, dy_dx, num_x)
```

where `dy_dx(i)` is an output array storing the constant derivative values and `num_x = 8` is the number of table entries.

The value and its derivative for any x ($0 \leq x \leq 1$) may obtained by a call to

```
call xy_table(x, xy_val, dy_dx, num_x, y, y_deriv)
```

where the output is the value of y and its derivative with respect to x. The value of an interpolant and its derivative between `xy_val(1,i)` and `xy_val(1,i+1)` are computed from

```
y       = xy_val(2,i-1) + (x - xy_val(1,i-1))*dy_dx(i)
y_deriv = dy_dx(i)
```

The precomputation of the `dy_dx(i)` avoids unnecessary numerical operations (especially the divide!).

Chapter 7

Adding a user solver

ADDING USER SOLVERS

There are several public domain linear equation solution routines available at various internet locations. Examples are *SuperLU*, *umfpack*, *Pardiso* to name three. To access any of these solvers it is necessary to add user modules named `umacr1.f` and `usolve.f` to *FEAP*. The module `umacrx.f` (x ranges between 0 and 9) has the basic form

```
subroutine umacr1(lct,ctl,prt)

include  'setups.h'     ! for parameter 'solver'
include  'umac1.h'      ! for parameter 'uct'

logical            :: prt
character (len=15) :: lct
real      (kind=8) :: ctl(3)

if(pcomp(uct,'mac1',4)) then
  uct = 'name'   ! Set name of command for solver

else
  if(pcomp(lct,'off',3)) then
    solver = .true.   ! Sets flag for FEAP solvers
      ... any other statements needed
  else
    solver = .false.  ! Sets flag for user solver
      ... any other statements needed
  endif
endif

end
```

and the module `usolve.f`

```
      subroutine usolve(flags,b)

c-----[--.----+----.----+----.-----------------------------------------]
c     Purpose:  Solver interface for SuperLU
c     Inputs:
c        flags(1) - Allocation and/or initialization phase
c        flags(2) - Perform factorization for direct solutions
c        flags(3) - Coefficient array unsymmetric
c        flags(4) - Solve equations
c        flags(5) - Purge storage of pointers
c        b(*)     - RHS vector
c     Outputs:
c        flags(5) - True if error occurs (for factor/solve only)
c-----[--.----+----.----+----.-----------------------------------------]
      implicit  none
      logical       :: flags(*)
      real (kind=8) :: b(*)

c     Presolve setups
      if(flags(1)) then
        ...

c     Solution steps for assembled equations

      else

c       Factor equations
        if(flags(2)) then
           ...
        endif

c       Perform solve
        if(flags(4)) then
           ...
        endif

c       Purge storage in 'factor'
        if(flags(5)) then
           ...
        endif
      endif
      end
```

Appendix A

Example: 2-Node Truss Element

An element routine carries out tasks according to the value assigned to the parameter `isw` as indicated in Table 5.2 To describe basic steps to program the various tasks defined by `isw`, we consider next the problem of a 2-node, linear elastic truss element for small deformation applications. The element is described in sufficient generality to permit solution of both two and three dimensional truss problems.

A.1 Linear truss element

The governing equations for a typical truss member element, shown in Figure A.1, are the balance of momentum equation:

$$\frac{\partial(A\sigma_{ss})}{\partial s} \; + \; A\, b_s = \rho\, A\, \ddot{u}_s$$

the strain-displacement equation for small deformations:

$$\epsilon_{ss} = \frac{\partial u_s}{\partial s}$$

and a constitutive equation. For example, considering a linear elastic material the constitutive equation may be written as

$$\sigma_{ss} = E\, \epsilon_{ss} \ .$$

Boundary and initial conditions must also be specified to obtain a well posed problem; however, our emphasis here is the derivation of the element arrays associated with the above differential equations. In the above:

- s is the coordinate along the truss member axis,
- b_s is a loading in direction s per unit length,
- A is the truss cross-section area,
- ρ is the mass density per unit volume,
- u_s is a displacement in direction s,
- $\dot{v}_s$ is an acceleration in direction s $(v = \dot{u})$,
- ϵ_{ss} is a strain along the truss member axis, and
- σ_{ss} is the stress on a truss cross section.

The equations may also be deduced from the variational equation

$$\delta\Pi \;=\; \int_L \delta\epsilon_{ss}\,\sigma_{ss}\,A\,ds \;+\; \sum_{i=1}^{d} \int_L \delta u_i\,\rho\,A\,\dot{v}_i\,ds \;-\; \sum_{i=1}^{d} \int_L \delta u_i\,b_i\,ds \;+\; \delta\Pi_{ext}$$

where $\delta\Pi_{ext}$ contains the boundary and loading terms not associated with an element. Where, in addition to previously defined quantities, we define:

- d is the spatial dimension of the truss (1, 2, or 3),
- x_i are the Cartesian coordinates in the d directions.
- L is the length of the truss member,
- δu_i is a virtual displacement in direction x_i,
- $\dot{v}_i$ is an acceleration in direction x_i $(v = \dot{u})$,
- b_i is a loading in direction x_i per unit length, and
- $\delta\epsilon_{ss}$ is a virtual strain along the truss axis.

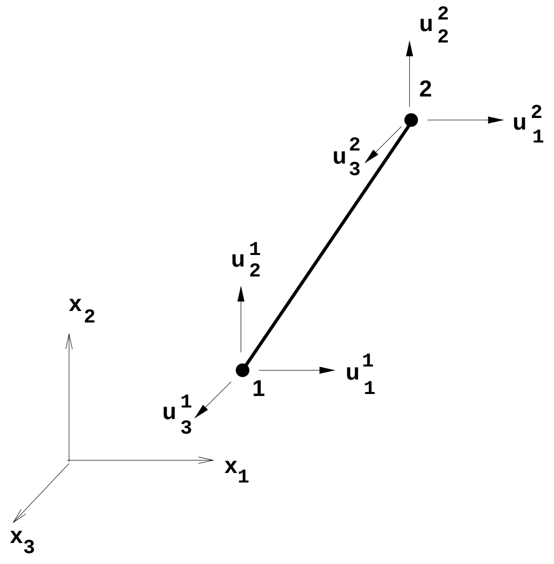


Figure A.1: 2-Node Truss Element

For a straight truss member the displacement along the axis, u_s may be expressed in terms of the components in the directions x_i as

$$u_s \;=\; \mathbf{l}\cdot\mathbf{u}(\,s\,,\,t\,) \;=\; \sum_{i=1}^{d} l_i\, u_i(\,s\,,\,t\,)$$

where t is time, $\mathbf{u}$ is the displacement vector with components u_i, $\mathbf{l}$ is a unit vector along the axis of the member with direction cosines l_i defined by

$$l_i = \frac{\partial x_i}{\partial s} = \frac{x_{i2} - x_{i1}}{L}$$

$$L^2 = \sum_{i=1}^{d} (\,x_{i2} \;-\; x_{i1}\,)^2$$

and x_{i1}, x_{i2} are the coordinates of nodes 1 and 2, respectively. The displacement components are interpolated on the 2-node truss member as

$$u_i(\,s\,,\,t\,) = (\,1 \;-\; \xi\,)\,u_{i1}(\,t\,) \;+\; \xi\,u_{i2}(\,t\,) \;\;;\;\; \xi = \frac{s}{L}$$

in which u_{i1}, u_{i2} are the displacements at nodes 1 and 2. The virtual displacements are obtained from the above by replacing u_i by δu_i, etc. The truss strain is

$$\epsilon_{ss} = \frac{\partial u_s}{\partial s} = \sum_{i=1}^{d} l_i\,\frac{\partial u_i}{\partial s}\;.$$

Using the interpolations for the displacement components yields

$$\epsilon_{ss} = \frac{1}{L^2} \sum_{i=1}^{d} \Delta x_i \, \Delta u_i$$

where

$$\Delta x_i = x_{i2} - x_{i1} = l_i \, L$$

and

$$\Delta u_i = u_{i2} - u_{i1} \ .$$

Thus, in matrix form the strain is

$$\epsilon_{ss} = \frac{1}{L^2} \sum_{i=1}^{d} \begin{bmatrix} -\Delta x_i & \Delta x_i \end{bmatrix} \begin{bmatrix} u_{i1} \\ u_{i2} \end{bmatrix}$$

Using the above displacement interpolations, the variational equation for the truss may be expressed in matrix form as

$$\begin{aligned} \delta\Pi = \begin{bmatrix} \delta u_{i1} & \delta u_{i2} \end{bmatrix} \Bigg\{ \int_L \frac{1}{L^2} \begin{bmatrix} -\Delta x_i \\ \Delta x_i \end{bmatrix} \sigma_{ss} A ds + \int_L \begin{bmatrix} 1-\xi \\ \xi \end{bmatrix} \rho A \begin{bmatrix} 1-\xi & \xi \end{bmatrix} ds \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \\ - \int_L \begin{bmatrix} 1-\xi \\ \xi \end{bmatrix} b_i ds \Bigg\} + \delta\Pi_{ext} \ . \end{aligned}$$

FEAP constructs the finite element arrays from the element residuals which are obtained from the negative of the terms multiplying the nodal displacements. Accordingly,

$$\begin{aligned} \mathbf{R}_i = \begin{bmatrix} R_{i1} \\ R_{i2} \end{bmatrix} \ = \ \int_L \begin{bmatrix} 1 \ - \ \xi \\ \xi \end{bmatrix} b_i \, ds \\ - \ \int_L \frac{1}{L^2} \begin{bmatrix} -\Delta x_i \\ \Delta x_i \end{bmatrix} \sigma_{ss} \, A \, ds \ - \ \int_L \begin{bmatrix} 1 \ - \ \xi \\ \xi \end{bmatrix} \rho \, A \begin{bmatrix} 1 \ - \ \xi & \xi \end{bmatrix} \, ds \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \end{aligned}$$

is the residual for the i-coordinate direction. For constant properties and loading over an element length (note that for this case the stress will also be constant since strains are constant on the element), the above may be integrated to yield

$$\mathbf{R}_i = \begin{bmatrix} R_{i1} \\ R_{i2} \end{bmatrix} = \frac{1}{2} b_i \, L \begin{bmatrix} 1 \\ 1 \end{bmatrix} \ - \ \frac{\sigma_{ss} \, A}{L} \begin{bmatrix} -\Delta x_i \\ \Delta x_i \end{bmatrix} \ - \ \frac{\rho \, A \, L}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \ . \qquad \text{(A.1)}$$

For the present we assume the material model is a linear elastic in which the stress is related to strain through

$$\sigma_{ss} = E \, \epsilon_{ss}$$

where E is the Young's modulus.

Based on a linear elastic material, the term in the residual involving σ_{ss} may be written as

$$\frac{\sigma_{ss}\,A}{L}\begin{bmatrix}-\,\Delta x_i\\ \Delta x_i\end{bmatrix} \;=\; \frac{E\,A}{L^3}\begin{bmatrix}-\,\Delta x_i\\ \Delta x_i\end{bmatrix}\sum_{j=1}^{d}\begin{bmatrix}-\,\Delta x_j & \Delta x_j\end{bmatrix}\begin{bmatrix}u_{j1}\\ u_{j2}\end{bmatrix} .$$

For the linear elastic material, a stiffness matrix may be expressed as

$$\mathbf{K}_{ij} \;=\; \frac{E\,A}{L^3}\begin{bmatrix}-\,\Delta x_i\\ \Delta x_i\end{bmatrix}\begin{bmatrix}-\,\Delta x_j & \Delta x_j\end{bmatrix} = \begin{bmatrix}k_{ij} & -k_{ij}\\ -k_{ij} & k_{ij}\end{bmatrix}$$

where

$$k_{ij} \;=\; \frac{E\,A}{L^3}\,\Delta x_i\,\Delta x_j\ .$$

The residual may now be written using a stiffness and mass matrix as

$$\mathbf{R}_i = \begin{bmatrix}R_{i1}\\ R_{i2}\end{bmatrix} = \frac{1}{2}\,b_i\,L\begin{bmatrix}1\\ 1\end{bmatrix} \;-\; \sum_{j=1}^{d}\begin{bmatrix}k_{ij} & -k_{ij}\\ -k_{ij} & k_{ij}\end{bmatrix}\begin{bmatrix}u_{j1}\\ u_{j2}\end{bmatrix} \;-\; \begin{bmatrix}m_{11} & m_{12}\\ m_{21} & m_{22}\end{bmatrix}\begin{bmatrix}\ddot{u}_{i1}\\ \ddot{u}_{i2}\end{bmatrix} \qquad \text{(A.2)}$$

with

$$m_{11} = m_{22} = \frac{\rho\,A\,L}{3}\quad ;\quad m_{12} = m_{21} = \frac{\rho\,A\,L}{6}\ .$$

For non-linear material behavior the residual must be computed using Equation A.1 with the stress replaced by the value computed from the constitutive equation.

The integration method for nodal quantities is taken as Newmark's method described in Section 5.4. The residual and tangent matrix for a Newton type method are now available and may be inserted into **R** and **S** after noting that for the truss that the damping matrix **C** is zero. The residual may be programmed directly from Equation A.1 and an implementation using the two dimensional form `r(ndf,nen)` is shown in Figure A.2.

Similarly, using the results from Section 5.4, the tangent matrix for the truss may be programmed as indicated in Figures A.3 and A.4.

A.2 A Non-linear Theory for a Truss

A simple non-linear theory for a two or three dimensional truss which may undergo large displacements for which the strains remain small may be developed by defining the axial strain approximation in each member as

```
      if(isw.eq.3 .or. isw.eq.6) then

c       Compute element length

        L2= 0.0d0
        do i = 1,ndm
          L2 = L2 + (xl(i,2) - xl(i,1))**2
        end do
        L = sqrt(L2)

c       Compute strain-displacement matrix

        Lr  = 1.d0/L2
        eps = 0.0d0
        do i = 1,ndm
          bb(i,1) = -(xl(i,2) - xl(i,1))*Lr
          bb(i,2) = -bb(i,1)
          eps     =  eps + bb(i,2)*(ul(i,2,1) - ul(i,1,1))
        end do

c       Compute mass terms

        cmd = rhoA*L/3.0d0
        cmo = cmd*0.5d0

c       Form body/inertia force vector (dm = prop. ld.)

        sigA = EA*eps*L
        body = 0.5d0*L*dm
        do i = 1,ndm
          r(i,1) = body*d(6+i)   - bb(i,1)*sigA
     &           - cmd*ul(i,1,5) - cmo*ul(i,2,5)
          r(i,2) = body*d(6+i)   - bb(i,2)*sigA
     &           - cmo*ul(i,1,5) -  cmd*ul(i,2,5)
        end do
```

Figure A.2: Element residual for two node truss

```
      if(isw.eq.3) then

c       Compute element length

        L2= 0.0d0
        do i = 1,ndm
          L2 = L2 + (xl(i,2) - xl(i,1))**2
        end do
        L = sqrt(L2)

c       Form stiffness multiplier

        dd = ctan(1)*EA*L

c       Compute strain-displacement matrix

        Lr = 1.d0/L2
        do i = 1,ndm
          bb(i,1) = -(xl(i,2) - xl(i,1))*Lr
          bb(i,2) = -bb(i,1)
          db(i,1) =  dd*bb(i,1)
          db(i,2) = -db(i,1)
        end do
```

Figure A.3: Truss Tangent Matrix. Part 1

```
c     Compute stiffness terms (N.B. ndm < or = ndf)

      i1 = 0
      do ii = 1,2
        j1 = 0
        do jj = 1,2
          do i = 1,ndm
            do j = 1,ndm
              s(i+i1,j+j1) = db(i,ii)*bb(j,jj)
            end do
          end do
        j1 = j1 + ndf
        end do
        i1 = i1 + ndf
      end do

c     Compute mass terms and correct for inertial effects

      cmd = ctan(3)*rhoA*L/3.0d0
      cmo = cmd*0.5d0
      do i = 1,ndm
        j      = i + ndf
        s(i,i) = s(i,i) + cmd
        s(i,j) = s(i,j) + cmo
        s(j,i) = s(j,i) + cmo
        s(j,j) = s(j,j) + cmd
      end do
    endif
```

Figure A.4: Truss Tangent Matrix. Part 2

$$\epsilon_{ss} = \frac{\partial u_s}{\partial s} + \frac{1}{2} \sum_{j=1}^{d-1} \left(\frac{\partial u_{nj}}{\partial s} \right)^2$$

where u_{nj} is a displacement component normal to the axis of the member. The virtual strain from a linearization of the strain is given as

$$\delta\epsilon_{ss} = \frac{\partial \delta u_s}{\partial s} + \sum_{j=1}^{d-1} \left(\frac{\partial \delta u_{nj}}{\partial s} \right) \left(\frac{\partial u_{nj}}{\partial s} \right) .$$

An algorithm to define the two orthogonal unit vectors which are normal to the member may be constructed by taking

$$\mathbf{v} = \mathbf{e}_k$$

where k is a direction for which a minimum value of the direction cosine l_i exists (for a 2-dimensional problem defined in the x_1, x_2 plane $\mathbf{v}$ may be taken as $\mathbf{e}_3$). Now,

$$\mathbf{n}_1 = \frac{\mathbf{v} \times \mathbf{l}}{|\, \mathbf{v} \times \mathbf{l} \,|}$$

and

$$\mathbf{n}_2 = \mathbf{l} \times \mathbf{n}_1 .$$

Using these vectors the two normal components of the displacement are given by

$$u_{nj}(\, s \,,\, t \,) = \mathbf{n}_j \cdot \mathbf{u}(\, s \,,\, t \,) = \sum_{i=1}^{d} n_{ji} \, u_i(\, s \,,\, t \,)$$

and the derivative by

$$\frac{\partial u_{nj}}{\partial s} = \sum_{i=1}^{d} n_{ji} \frac{\partial u_i}{\partial s} .$$

Collecting terms and combining with previously defined quantities the virtual strain may be written as

$$\delta\epsilon_{ss} = \frac{\partial \delta \mathbf{u}}{\partial s} \cdot [\, \mathbf{g} \,]$$

where

$$\mathbf{g} = \mathbf{l} + \sum_{j=1}^{d-1} \frac{\partial u_{nj}}{\partial s} \mathbf{n}_j .$$

After differentiation of the displacement field the discrete form of the virtual strain is given by

$$\delta\epsilon_{ss} = \frac{1}{L} \begin{bmatrix} \delta\mathbf{u}_1 & \delta\mathbf{u}_2 \end{bmatrix} \cdot \begin{bmatrix} -\,\mathbf{g} \\ \mathbf{g} \end{bmatrix} .$$

Substituting the above virtual strain expression into the weak form gives the modified residual expression

$$\mathbf{R}_i = \frac{1}{2}\, b_i\, L \begin{bmatrix} 1 \\ 1 \end{bmatrix} \;-\; \sigma_{ss}\, A \begin{bmatrix} -\, g_i \\ g_i \end{bmatrix} \;-\; \rho\, A\, \frac{L}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \;. \tag{A.3}$$

The tangent tensor is obtained by linearizing the residual as shown previously. The only part which is different is the term with σ_{ss}. Noting that

$$d\epsilon_{ss} \;=\; \big[\, \mathbf{g} \big] \cdot \frac{\partial d\mathbf{u}}{\partial s}$$

and

$$d\, \delta\epsilon_{ss} = \frac{\partial \delta\mathbf{u}}{\partial s} \cdot (\mathbf{n}_1 \otimes \mathbf{n}_1 \;+\; \mathbf{n}_2 \otimes \mathbf{n}_2) \cdot \frac{\partial d\mathbf{u}}{\partial s} \;.$$

If the $\mathbf{n}_i$ are constructed as *column* vectors then the tensor product becomes a matrix defined as

$$\mathbf{G} \;=\; \mathbf{n}_1 \otimes \mathbf{n}_1 \;+\; \mathbf{n}_2 \otimes \mathbf{n}_2 \;=\; \mathbf{n}_1\, \mathbf{n}_1^T \;+\; \mathbf{n}_2\, \mathbf{n}_2^T \;.$$

With these definitions, the *tangent* matrix for the non-linear problem is given as

$$\mathbf{K}_{ij} \;=\; \frac{EA}{L} \begin{bmatrix} -\, g_i \\ g_i \end{bmatrix} \begin{bmatrix} -\, g_j & g_j \end{bmatrix} \;+\; \frac{\sigma_{ss}\, A}{L^2} \begin{bmatrix} G_{ij} & -\, G_{ij} \\ -\, G_{ij} & G_{ij} \end{bmatrix} \;.$$

Notice that for the linear problem

$$g_i \;=\; \frac{\Delta x_i}{L}$$

thus, the only difference between the linear and non-linear problem is the definition of ϵ_{ss} in terms of displacements, the modification for geometric effects for the g_i and the second term on the tangent matrix which is sometimes called the *geometric* stiffness part.

Appendix B

Compiling in C

User modules may be added in either Fortran or C by using proper variable types for each quantity. In Fortran variables are passed between modules either as arguments to the module or in `common` blocks. To facilitate variable typing common blocks are defined as `include` statements. In C these must be converted to structures.

The various variable types used in FEAP are shown in Table B.1.

Fortran Type	C Type	Description
`integer`	`int`	All variables except pointers
`integer (kind=8)`	`long int`	Array pointers
`real (kind=4)`	`float`	Some graphics variables
`real (kind=8)`	`double`	All floating point values

Table B.1: Fortran and C variable typing.

Bibliography

[1] R.L. Taylor and S. Govindjee. *FEAP - A Finite Element Analysis Program, User Manual*. University of California, Berkeley. http://projects.ce.berkeley.edu/feap.

[2] G.G. Weber, A.M. Lush, A. Zavaliangos, and L. Anand. An objective time-integration procedure for isotropic rate-independent and rate-dependent elastic-plastic constitutive equations. *International Journal of Plasticity*, 6:701–744, 1990.

[3] J.N. Lyness and C.B. Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4:202–210, 1967.

[4] J.N. Lyness. Numerical altorithms based on the theory of complex variables. In *Proceedings 22nd National Conference A.C.M.*, pages 125–133, 1967. Publication P-67.

[5] J.N. Lyness. Differentiation formulas for analytic functions. *Mathematics of Computation*, 22:352–362, 1968.

[6] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40:110–112, 1998.

[7] L.I. Cerviño and T.R. Bewley. On the extension of the complex-step derivative technique to pseudospectral algorithms. *Journal of Computational Physics*, 187:544–549, 2003.

[8] J. Kim, D.G. Bates, and I. Postlethwaite. Nonlinear robust performance analysis using complex-step gradient approximation. *Automatica*, 42:177–182, 2006.

[9] S. Kim, J. Ryu, and M. Cho. Numerically generated tangent stiffness matrices using the complex variable derivative method for nonlinear structural analysis. *Computer Methods in Applied Mechanics and Engineering*, 200:403–413, 2011.

[10] K.-L. Lai and J.L. Crassidis. Extensions of the first and second complex-step derivative approximations. *Journal of Computational and Applied Mathematics*, 219:276–293, 2008.

[11] M. Tanaka, M. Fujikawa, D. Balzani, and J. Schröder. Robust numerical calculation of tangent moduli at finite strains based on complex-step derivative approximation and its application to localization analysis. *Computer Methods in Applied Mechanics and Engineering*, 269:454–470, 2014.

[12] R. Kran and K. Khandelwal. Complex step derivative approximations for numerical evaluation of tangent moduli. *Computers and Structures*, 140:1–13, 2014.

[13] S. Govindjee, J. Strain, T.J. Mitchell, and R.L. Taylor. Convergence of an efficient local least-squares fitting method for bases with compact support. *Computer Methods in Applied Mechanics and Engineering*, 213–216:84–92, 2012. http://dx.doi.org/10.1016/j.cma.2011.11.017.

[14] T.J. Mitchell, S. Govindjee, and R.L. Taylor. A method for enforcement of Dirichlet boundary conditions in isogeometric analysis. In Dana Mueller-Hoeppe, Stefan Loehnert, and Stefanie Reese, editors, *Recent Developments and Innovative Applications in Computational Mechanics*, pages 283–293. Springer-Verlag, Berlin Heidelberg, 2010.

[15] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method*, volume 1. McGraw-Hill, London, 4^{th} edition, 1989.

[16] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method: The Basis*, volume 1. Butterworth-Heinemann, Oxford, 5^{th} edition, 2000.

[17] O.C. Zienkiewicz, R.L. Taylor, and J.Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals*. Elsevier, Oxford, 6^{th} edition, 2005.

[18] O.C. Zienkiewicz, R.L. Taylor, and J.Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals*. Elsevier, Oxford, 7^{th} edition, 2013.

[19] M. Abramowitz and I.A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publications, New York, 1965.

[20] T.J.R. Hughes. *The Finite Element Method: Linear Static and Dynamic Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1987.

[21] M. Gellert and R. Harbord. Moderate degree cubature formulas for 3-D tetrahedral finite-element approximations. *Communications in Applied Numerical Methods*, 7:487–495, 1991.

Index