

FEAP - - A Finite Element Analysis Program

Version 7.3 Programmer Manual

Robert L. Taylor
Department of Civil and Environmental Engineering
University of California at Berkeley
Berkeley, California 94720-1710
E-Mail: rlt@ce.berkeley.edu

December 2000

Contents

1	Introduction	1
1.1	Setting Program Capacity and Options	1
1.2	Uses of Common and Include Statements	3
2	Data Input and Output	5
2.1	Parameters and Expressions	5
2.2	Array Outputs	7
3	Allocating Arrays	8
4	User Functions	13
4.1	User Mesh Input Functions.	13
4.2	User Solution Command Functions.	14
4.3	User Material Models	15
4.3.1	The UCONST Module	16
4.3.2	The UMODEL Module	16
5	Adding Elements	20
5.1	Material property storage	21
5.2	Non-linear Transient Solution Forms	22
5.3	Example: 2-Node Truss Element	34
5.3.1	Theory for a Truss	35
5.4	Additional Options in Elements	42
5.4.1	Task 1 Options	42
5.4.2	Task 3 Options	42
5.5	Projection of element variables to nodes	43
5.6	Elements with History Variables	44
5.6.1	Assigning amount of storage for each element	46
5.6.2	Accessing history data for each element	46
5.7	Energy Computation	47
5.8	A Non-linear Theory for a Truss	48
6	Utility routines	51

6.1	Numerical quadrature routines	51
6.1.1	One dimensional quadrature	51
6.1.2	Two dimensional quadrature	52
6.1.3	Three dimensional quadrature	53
6.2	Shape function subprograms	54
6.2.1	Shape functions in one-dimension	54
6.2.2	Shape functions in two-dimensions	55
6.2.3	Shape functions in three-dimensions	57
6.3	Eigenvalues for 3×3 matrix	57
6.4	Mesh plot routines	58

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 Capacity and Options

The size of problems which may be solved by *FEAP* depends on the amount of memory provided in blank common as well as solution options used. The capacity of blank common is set in the main program module FEAP73 in the section between the comments labeled START SOLUTION SETS and END SOLUTION SETS. To change the size of the blank common array reassign the value as:

```
parameter ( mmax = 1000000 )
```

where in the above the size of blank common is set to one million *integer words* of storage. If real arrays are double precision there are only a half million (500000) words of real data available. The size of the parameter may not be made arbitrarily large and depends on system configurations for swap and disk space.

The IPR parameter in the FEAP73 module controls the specification of the size of REAL variables. For typical UNIX and PC systems all real variables should be double precision and IPR is set to 2. For systems in which REAL*8 variables are *single precision* with the same work length as integer variables 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 to single precision unless a careful assessment of compiler behavior for `REAL*8` variables has been made.

By placing an alphanumeric version of each manual page in a separate file which has the name of the command and a `.t` extender (e.g. `coor.t` for the mesh coordinate input command) it is possible to read each page during execution using the `HELP,name` command (where `name` is the command name whose manual page is to be read). For this option to work properly it is necessary to define the path name to each manual page in the `FEAP73` module. For example:

```
file(1) = 'c:\Feap\Manual\Mesh/'
file(2) = 'c:\Feap\Manual\Macr/'
file(3) = 'c:\Feap\Manual\Plot/'
```

defines a typical path for a PC system. Each system requires a proper path definition. *FEAP* will add the requested command name to each of the above paths to find mesh, solution, or plot commands.

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 `FEAP73` 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.

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 set in the `FEAP73` 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 or 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** command. The `PSREVS` variable reverses the color sequence.

Arrays in *FEAP* may be dynamically allocated during execution. Thus, it is possible to define and destroy arrays as well as to increase or decrease the size of an array. A parameter is provided to control when an array is to be decreased in size - which causes all arrays at higher locations in the blank common to be moved to fill the decrease. The parameter is `INCR` and an array is decreased in size only when the new size is less than the old size by the assigned value.

The last parameter which may be set in the **FEAP73** 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

```
logical          debug
common /debugs/ 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
c    and/or
        write( *,*) 'LABEL',list ... ! writes to screen
      endif ! debug
```

This device supplements use of available debuggers on the computer.

1.2 Uses of Common and Include Statements

FEAP contains many **COMMON** statements which 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`. The only exception is for the blank common which uses the file name `comblk.h`. All include files are located in the directory `include`.

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 255 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(.....)
logical errck

integer          ior,iow
common /iofile /ior,iow

real*8 td(5)

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

The parameters defined in the common block are:

```
ior   - input file unit number (if negative, input
```

```

        from keyboard)
iow   - output 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*4  t
integer j
logical errck

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 255 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 text(16)*16
real*8   td(16)

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

```

The value of either the 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*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 `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.

Chapter 3

ALLOCATING ARRAYS

The blank common in *FEAP* is defined in the form

```
real*8    hr
integer    mr
common    hr(1),mr(1000)
```

and placed in a file `comblk.h` in the `INCLUDE` directory. Thus, during any solution the `hr` array will overlay the `mr` array, that is, for `REAL*8` variables combined with `INTEGER` variables, `hr(2)` uses the same memory location as `mr(1)` and `mr(2)`. This mechanism permits references to positions in `hr` beyond 1 and `mr` beyond 1000 if *FEAP* is compiled without strict array bound checking. The actual length of the blank common also does not need to be given explicitly in each routine as it is assigned by the parameter statement in the main program `FEAP73` module. Generally, there the common is specified with a very large value, for example,

```
integer    mr
common    mr(10000000)
```

or more is recommended. For some compilers index calculations are different for small and large sized arrays, thus, it is recommended that some value greater than 1 (e.g., the 1000 shown above) be used to protect against incorrect array index computations. Generally, routines will use the include file as

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

so that no special changes are necessary.

Using the above 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* and a subprogram `UALLOC` permits users to add their own arrays which are to be allocated from blank common. The basic use of the routines is provided by an instruction

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

or

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

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 space in blank common so that no conflicts occur between references to `hr` and `mr` arrays. Access of information in each of the arrays is performed using a pointer which for `PALLOC` is in

```
integer          np
common /pointer/ np(500)
```

and for `UALLOC` is in

```
integer          up
common /upointer/ up(200)
```

These commons are saved in the include files `pointer.h` and `upointer.h`, respectively.

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 an include file `cdata.h`. The new arrays re defined using the temporary names `TEMP1` and `TEMP2` which have numerical locations '111' and '112', respectively. 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:

NAME	Num.	dim 1	dim 2	dim 3	Description
ANG	45	numnp	-	-	Angle
D	25	250	nummat	-	Material parameters
F	27	ndf	numnp	2	Force and Displacement
ID	31	ndf	numnp	2	Equation nos.
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

```

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 be 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` execution command.

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

```

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

```

NAME	Num.	dim 1	dim 2	dim 3	Description
ANGL	46	nen	-	-	Angle
LD	35	nst	-	-	Assembly nos.
P	35	nst	-	-	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

are given. Use of these statements results in the pointers `np(111)` and `np(112)` being set to zero and the space in the blank common re. If any arrays have been allocated subsequent to defining `TEMP1` and `TEMP2` their values are moved up in the blank common and their pointers are redefined. In this way no ‘memory leaks’ should occur and unneeded memory space can be released.

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. Note that an array will not have its size decreased unless it differs by more than the value specified for the variable `INCRD` in the main program module `FEAP73`.

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 blank common 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 add new solution commands, or to add new plot capabilities.

4.1 User Mesh Input Functions.

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:

```
subroutine umesh1(uprt)

c   User defined routine to input mesh data to FEAP

implicit none

include 'umac1.h' ! Contains UCT variable
logical uprt

c   Set name 'mes1' to user defined

if(pcomp(uct,'mes1',4)) then
    uct = 'xxxx' ! Set user defined command name
else

c   User execution function statements follow

end if

end
```

The parameter `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` for the name of the command. The default names are `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 `MANUa1,3` and then the `HELP` command).

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.

4.2 User Solution Command Functions.

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.

```

subroutine umacr0(lct,ctl,uprt)

c   User macro statement function

implicit none

include 'umac1.h'      ! Contains the variable UCT

logical uprt
character lct*15
real*8    ctl(3)

c   Set command word

if(pcomp(uct,'mac0',4)) then
    uct = 'xxxx'
else

c   User command statements are placed here

endif

```

```
end
```

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 instead of the default value of `mac0`.

4.3 User Material Models

Users may add material models to elements by appending subprograms `UCONST` and `UMODEL` to the *FEAP* system. The subprogram `UCONST` defines parameters used by the model as well as the model *number*. The subprogram `UMODEL` 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.

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 ...
```

The role of the `xxxx` and `vi` data will be described in Section 4.3.1.

It is possible to use standard input parameters defined in Tables 5.5 to 5.8, as well as 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 ...
```

No standard commands should follow the `UCON` command.

Alternatively, users may input elastic properties as part of their `UCONST` module. 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.

4.3.1 The UCONST Module

A sample module for a user constitutive model is shown in Fig. 4.1. As shown in this figure, the UCONST module has 6 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 UCONST module using the routines PINPUT and TINPUT described in Sect. 2.1. The values of user parameters must be saved in the array **ud(*)** (the third argument of UCONST). In the current version there are 50 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 UMODEL module.

4.3.2 The UMODEL Module

A sample for the MODEL module is shown in Fig. 4.2. This subprogram will be called by many of the elements included within FEAP if a user model has been specified as part of the MATE mesh data (see previous subsection). The user mmodel will not be called for truss, frame, plate, and shell elements which use resultant models to describe behavior. Also, and form which requires a one-dimensional model will not use the UMODEL 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. Strains are passed to UMODEL in the argument array **eps(6)** and stored in the order

$$\boldsymbol{\epsilon} = [\epsilon_{11} \quad \epsilon_{22} \quad \epsilon_{33} \quad \gamma_{12} \quad \gamma_{23} \quad \gamma_{31}]^T$$

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 **assig(6)** and **dd(6,6)**, respectively. All values are to double precision (i.e., REAL*8).

When UMODEL is called the material model to be computed will be assigned to the parameter **umat**, which is as defined in the module UCONST. 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 5.8 and the array `ud(*)` contains values as assigned in the user module `UCONST`.

For constitutive equations which have additional (internal) variables which evolve with time users must define entries for the `h1(*)` array. The number of entries available in the array for *each point* is `nhv` and will be the value assigned to `n1` in module `UCONST`. Values from the previous time step are passed back to the module in the array `hn(*)` (which also contains `nhv` 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`).

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 umodel(umat,eps,theta,td,d,ud,hn,h1,nh, sig,dd, isw)
c-----[--.----+----.----+----.-----]
c      Purpose: User Constitutive Model

c      Input:
c          umat   - User material type
c          eps(*) - Current strains at point      (small deformation)
c               - Deformation gradient at point (finite deformation)
c          theta  - Trace of strain at point
c               - Determinant of deformation gradient
c          td     - Temperature change
c          d(*)   - Program material parameters (nnd)
c          ud(*)  - User material parameters (nud)
c          hn(nh) - History terms at point: t_n
c          h1(nh) - History terms at point: t_n+1
c          nh    - Number of history terms
c          isw   - Solution option from element

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

c-----[--.----+----.----+----.-----]
      implicit none

      integer umat,nh,isw, i
      real*8  td
      real*8  eps(*),theta(*),d(*),ud(*),hn(nh),h1(nh), sig(6),dd(6,6)

c      Material Model 1

      if(umat.eq.1) then

c          Dummy model: sig = d(1)*eps

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

      endif

      end

```

Figure 4.2: Sample UCONST module

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. Each element subprogram must be added before loading the *FEAP* library since dummy subprograms are included in the library to avoid unsatisfied externals. The basic structure for an element routine is shown in Figures 5.1 and 5.2.

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 5.1¹. Some of the options for additional data passed through common blocks is shown in Figure 5.3 with each variable defined in Table 5.2. 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.

Each element can carry out tasks according the value of the task parameter **isw**. A list of currently available tasks is shown in Tables 5.2 and 5.3. It is not necessary that all options be coded. However, to use the features available in *FEAP* it is necessary to program at least the tasks 0 to 6, 8, and 10. If elements have variables which need to be saved between time steps history variables may be defined as described in Section 5.6 and tasks 12 and 14 may be necessary. Finally, if special plotting options are desired

¹Note in Table 5.1 that *FEAP* transfers the values for most of the solution parameters 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.

it may be necessary to program task 20 (note that contours for element variables such as stress, strain, etc. are developed from task 8).

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 INPT2D. This subroutine is accessed by the statement:

```
CALL INPT2D(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 the element type. 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 the following tables.

```

subroutine elmtnn(d,ul,xl,ix,tl,s,r,ndf,ndm,nst,isw)
c   Prototype FEAP Element Routine:  nn = 01 to 50
implicit none
c   Common blocks:  See Figure 5.2.
integer ndf,ndm,nst,isw
integer ix(nen1,1)
real*8  d(*),ul(ndf,*),xl(ndm*),tl(*),s(nst,nst),r(nst)

if(isw.eq.0 .and. ior.lt.0) then
c   Output element description
write(*,*) ' Elmt 1: Element description'

elseif(isw.eq.1) then
c   Input/output of property data after command: 'mate'
c   d(*) stores information for each material set
c   Return: nh1 = number of nh1/nh2 words/element
c   Return: nh3 = number of nh3      words/element

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

```

Figure 5.1: FEAP Element Subprogram. Part 1

```

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

    elseif(isw.eq.4) then
c      Output element quantities (e.g., stresses)
    elseif(isw.eq.5) then
c      Return: Element mass matrix
c      s(nst,nst) consistent matrix
c      r(ndf,nen) diagonal matrix

    elseif(isw.eq.6) then
c      Compute residual only
c      r(ndf,nen) element residual

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

    elseif(isw.eq.8) then
c      Compute stress projections to nodes (diagonal)
c      p(nen)    projection weight: wt(nen)
c      s(nen,*)  projection values: st(nen,*)
c                (default: project 8 quantities)
    endif
end

```

Figure 5.2: *FEAP* Element Subprogram. Part 2

5.2 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

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

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

```

character*4    o,head
common /bdata/ o,head(20)

integer       numnp,numel,nummat,nen,neq,ipr
common /cdata/ numnp,numel,nummat,nen,neq,ipr

integer       nstep,niter,naugm, titer,taugm,  iaugm,  iform
common /counts/ nstep,niter,naugm, titer,taugm,  iaugm,  iform

real*8        dm
integer       n,ma,mct,iel,nel
common /eldata/ dm,n,ma,mct,iel,nel

real*8        tt
common /elplot/ tt(200)

real*8        bpr,  ctan
common /eltran/ bpr(3),ctan(3)

real*8        ut
common /eluser/ ut(200)

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

integer       ior,iow
common /iofile/ ior,iow

integer       neph
real*8        erav,j_int
common /prstrs/ neph,ner,erav,j_int(3)

integer       ndf,ndm,nen1,nst,nneq
common /sdata/ ndf,ndm,nen1,nst,nneq

real*8        ttim,dt,c1,c2,c3,c4,c5
common /tdata/ ttim,dt,c1,c2,c3,c4,c5

real*8        hr
integer       mr
common /      / hr(1),mr(1000)

```

Figure 5.3: *FEAP* Element Common Blocks

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

$$\mathbf{R}_i^{(k+1)} = \mathbf{R}_i^{(k)} + \frac{\partial \mathbf{R}_i}{\partial \alpha_j} \Big|^{(k)} d\alpha_j^{(k)} = \mathbf{0}$$

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

$$\mathbf{S}_{ij}^{(k)} = - \frac{\partial \mathbf{R}_i}{\partial \alpha_j} \Big|^{(k)}$$

and solve

$$\mathbf{S}_{ij}^{(k)} d\alpha_j^{(k)} = \mathbf{R}_i^{(k)}$$

The solution is updated using

$$\alpha_j^{(k+1)} = \alpha_j^{(k)} + d\alpha_j^{(k)}$$

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

$$\alpha_j^{(0)}(t_{n+1}) = \alpha_j(t_n)$$

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).

```
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

Variable	Definition
o	Page eject option
head	Title record
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
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
dm	Element proportional load
n	Current element number
ma	Current element material set
mct	Print counter
iel	User element number
nel	Number nodes on current element
tt	Element stress values for TPL0t
bpr	Principal stretch
ctan	Element multipliers
ut	Element user values for TPL0t

Table 5.2: *FEAP* common block definitions

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

Variable	Definition
nh1	Pointer to t_n history data
nh2	Pointer to t_{n+1} history data
nh3	Pointer to element history
ior	Current input logical unit
iow	Current output logical unit
nph	Pointer to global projection arrays
ner	Pointer to global error indicator
erav	Element error value
j-int	J integral values
ndf	Maximum dof/node
ndm	Mesh space dimension
nen1	Dimension 1 on IX array
nst	Size of element matrix
nneq	Total dof in problem
ttim	Current time
dt	Current time increment
ci	Integration parameters
hr	Real array data
mr	Integer array data

Table 5.3: *FEAP* common block definitions

of the displacements (i.e., α_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

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

in which δ_{kj} is the Kronecker 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.

In *FEAP* the element tangent matrix, \mathbf{S}_{ij} , is stored as a two dimensional array which is dimensioned as $\mathbf{s}(\mathbf{nst}, \mathbf{nst})$, where \mathbf{nst} is the product of \mathbf{ndf} and \mathbf{nen} , with \mathbf{ndf} the *maximum number of degree-of-freedom at any node in the problem* and \mathbf{nen} the maximum number of nodes on any element. The ordering of the unknowns into \mathbf{nst} must be carefully aligned in order for *FEAP* to properly assemble each element matrix into the global tangent. The ordering is such that sub-matrices are defined for each

Task	Description	Access Command
0	Output label	SHOW, ELEM
1	Input $\mathbf{d}(\ast)$ parameters	Mesh:MATE, n
2	Check elements	Soln:CHECK
3	Compute tangent/residual Store in \mathbf{S}/\mathbf{r}	Soln:TANG UTAN
4	Output element variables	Soln:STRE
5	Compute cons/lump mass Store in \mathbf{S}/\mathbf{r}	Soln:MASS MASS, LUMP
6	Compute residual	Soln:FORM, REAC Plot:REAC
7	Surface load/tangents	Mesh:SLOAD
8	Nodal projections	Soln:STRE, NODE Plot:STRE, PSTR
9	Damping	Soln:DAMP
10	Augmented Lagrangian update	Soln:AUGM
11	Error estimator	Soln:ERRO
12	History update	Soln:TIME
13	Energy/momentum	Soln:TPLO, ENER
14	Initialize history	BATCH, INTER
15	Body force	Mesh:BODY
16	J integrals	Soln: JINT
17	Set after activation	Soln:ACTI
18	Set after deactivation	Soln:DEAC
19	Not available: used in modal	–
20	Element plotting	Plot:PELM

Table 5.4: Task Options for *FEAP* Element Subprogram

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
6	-	Quadrature order for outputs
7	a	Mass interpolator ($a = 0$: Diagonal; $a = 1$: Consistent)
8	q	Loading intensity (plates/shells)
9	T_0	Stress free reference temperature
10	κ	Shear factor (plates/shells/beams)
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
17	etype	Element formulation: 1 - displ; 2 - mixed; 3 - enhanced
18	dtype	Deformation type: <: finite; > small
19	tdof	Thermal degree-of-freedom
20	imat	Non-linear elastic material type
21	d_{11}	Material moduli
22	d_{22}	Material moduli
23	d_{33}	Material moduli
24	d_{12}	Material moduli
25	d_{23}	Material moduli
26	d_{31}	Material moduli
27	g_{12}	Material moduli
28	g_{23}	Material moduli
29	g_{31}	Material moduli
30	htype	Heat flag

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 plate
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 5.6: 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	-	Unused
68	-	Unused
69	-	Rotational mass factor
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	-	Unused
80		Method: Type 1
81		Method: Type 2
82	-	Unused
83	-	Unused
84	-	Unused
85	-	Unused
86	-	Unused
87	-	Unused
88	-	Unused
89	-	Unused
90	d_{31}	Plane stress recovery

Table 5.7: Material Parameters

Parameter	Name	Description
91	d_{32}	Plane stress recovery
92	α_3	Plane stress recovery
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	-	Unused
128	-	Unused
129	-	Unused
130	nseg	Number of hardening segments
131-148	-	Segment data sets $e_p Y_{iso} H_{kin}$
149	-	Unused
150	-	Piezoelectric flag
151-159	-	Piezoelectric data
160	-	Initial stress flag
161-166	σ_{ij}	Initial stresses (constant)
167	-	Tension/compression only indicator
170	C	Fung pseudoelastic model modulus
171	a_1	Fung model energy parameter
172	a_2	Fung model energy parameter
173	a_3	Fung model energy parameter
174	a_4	Fung model energy parameter
175	a_5	Fung model energy parameter
176	a_6	Fung model energy parameter
177	a_7	Fung model energy parameter
178	a_8	Fung model energy parameter
179	a_9	Fung model energy parameter
180-200	-	Unused

Table 5.8: Material Parameters

node attached to the element. Thus

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

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,

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

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

In *FEAP*, the element residual may be stored as 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

$$\mathbf{R}_i = \begin{bmatrix} R_1^i \\ R_2^i \\ R_3^i \\ \vdots \\ R_{ndf}^i \end{bmatrix}$$

The two dimensional form places the entries \mathbf{R}_i as columns. Accordingly,

$$\mathbf{R} = [\mathbf{R}_1 \quad \mathbf{R}_2 \quad \mathbf{R}_3 \quad \cdots]$$

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).

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

Table 5.9: Tangent Parameters

The arrays $x1(i, j)$, $ul(i, j, 1)$, $ul(i, j, 4)$ and $ul(i, j, 5)$ (described in Table 5.1) are used to obtain the nodal coordinates, displacements, velocities and accelerations, respectively. When *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 \mathbf{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 \mathbf{s} for the various cases, a parameter array $\mathbf{ctan}(3)$ is passed to the subprogram in labeled common \mathbf{eltran} . When the task parameter isw is 3, the values in the \mathbf{ctan} array are interpreted according to Table 5.9.

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 \mathbf{ctan} array are set by *FEAP* according to the active transient solution option. For a static 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.

5.3 Example: 2-Node Truss Element

An element routine carries out tasks according to the value assigned to the parameter isw as indicated in Table 5.4 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.

5.3.1 Theory for a Truss

The governing equations for a typical truss member element, shown in Figure 5.5, are the balance of momentum equation:

$$\frac{\partial(A\sigma_{ss})}{\partial s} + Ab_s = \rho A \ddot{u}_s$$

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

$$\delta\Pi = \int_L \delta\epsilon_{ss} \sigma_{ss} A ds + \sum_{i=1}^d \int_L \delta u_i \rho A \dot{v}_i ds - \sum_{i=1}^d \int_L \delta u_i b_i ds + \delta\Pi_{ext}$$

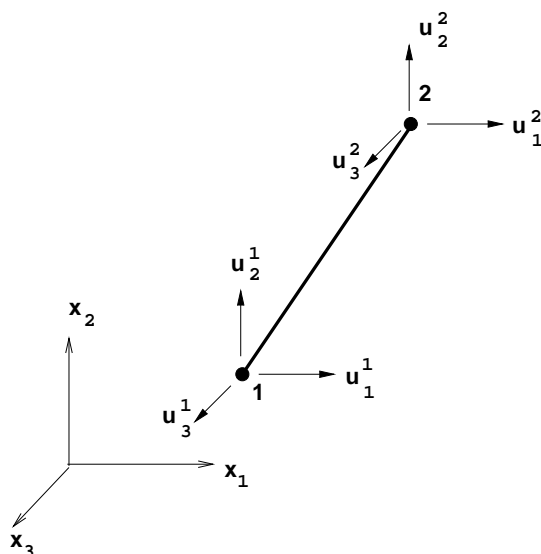


Figure 5.5: 2-Node Truss Element

$\delta\Pi_{ext}$ 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.

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

$$u_i(s, t) = (1 - \xi) u_{i1}(t) + \xi u_{i2}(t) ; \quad \xi = \frac{s}{L}$$

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

$$\begin{aligned} \delta \Pi = & [\delta u_{i1} \quad \delta u_{i2}] \left\{ \int_L \frac{1}{L^2} \begin{bmatrix} -\Delta x_i \\ \Delta x_i \end{bmatrix} \sigma_{ss} A ds + \int_L \begin{bmatrix} 1 - \xi \\ \xi \end{bmatrix} \rho A \begin{bmatrix} 1 - \xi & \xi \end{bmatrix} ds \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \right. \\ & \left. - \int_L \begin{bmatrix} 1 - \xi \\ \xi \end{bmatrix} b_i ds \right\} + \delta \Pi_{ext} \end{aligned}$$

FEAP constructs the finite element arrays from the element residuals which are obtained from the negative of the terms multiplying the nodal displacements. Accordingly,

$$\mathbf{R}_i = \begin{bmatrix} R_{i1} \\ R_{i2} \end{bmatrix} = \int_L \begin{bmatrix} 1 - \xi \\ \xi \end{bmatrix} b_i ds$$

$$- \int_L \frac{1}{L^2} \begin{bmatrix} -\Delta x_i \\ \Delta x_i \end{bmatrix} \sigma_{ss} A ds - \int_L \begin{bmatrix} 1 & -\xi \\ \xi & \end{bmatrix} \rho A [1 \quad -\xi \quad \xi] ds \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix}$$

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

$$\mathbf{R}_i = \begin{bmatrix} R_{i1} \\ R_{i2} \end{bmatrix} = \frac{1}{2} b_i L \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \frac{\sigma_{ss} A}{L} \begin{bmatrix} -\Delta x_i \\ \Delta x_i \end{bmatrix} - \frac{\rho A L}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \quad (5.1)$$

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

$$\mathbf{R}_i = \begin{bmatrix} R_{i1} \\ R_{i2} \end{bmatrix} = \frac{1}{2} b_i L \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \sum_{j=1}^d \begin{bmatrix} k_{ij} & -k_{ij} \\ -k_{ij} & k_{ij} \end{bmatrix} \begin{bmatrix} u_{j1} \\ u_{j2} \end{bmatrix} - \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \quad (5.2)$$

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 5.1 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 5.2. The residual and tangent matrix for a Newton type method are now available and may be inserted into \mathbf{R} and \mathbf{S} after noting that for the truss that the damping matrix \mathbf{C} is zero. The residual may be programmed directly from Equation 5.1 and an implementation using the two dimensional form $\mathbf{r}(\mathbf{ndf}, \mathbf{nen})$ is shown in Figure 5.6.

Similarly, using the results from Section 5.2, the tangent matrix for the truss may be programmed as indicated in Figures 5.7 and 5.8.

```

if(isw.eq.3 .or. isw.eq.6) then

c    Compute element length

    L2= 0.0d0
    do i = 1,ndm
      L2 = L2 + (x1(i,2) - x1(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) = -(x1(i,2) - x1(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 5.6: Element residual for two node truss

```
    if(isw.eq.3) then
c      Compute element length
      L2= 0.0d0
      do i = 1,ndm
        L2 = L2 + (x1(i,2) - x1(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) = -(x1(i,2) - x1(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 5.7: 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
  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 5.8: Truss Tangent Matrix. Part 2

5.4 Additional Options in Elements

FEAP permits some additional options to be included within element tasks.

5.4.1 Task 1 Options

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 Section 5.3, 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-freedom 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 any unneeded degrees-of-freedom.

```
do i = ndm+1,ndf
  ix(i) = 0
end do ! i
```

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.

Utility routines are also provided to assist users in providing the necessary list of nodes needed to properly draw the mesh each element type during plot outputs. The names of the routines are listed in Table 5.10 and each routine is called as

```
call pname (iel)
```

where `iel` is an integer parameter defined in common `eldata`. If no call to a subprogram is included each element is assumed to be a 4 to 9 node quadrilateral and default drawing order will be assigned. Users may construct their own drawing order also by following the steps employed in one of the routines defined in Table 5.10.

5.4.2 Task 3 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

Routine Name	Description
PLTLN2	2-node line element
PLTRI3	3-node triangular element
PLQUD4	4-node quadrilateral element
PLTRI6	6-node triangular element
PLTET4	4-node tetrahedron element
PLBRK8	8-node brick element

Table 5.10: Element Plot Definition Subprograms

```
real*8          tt
common /elplot/ tt(100)
```

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
```

5.5 Projection of element variables to nodes

The `STRESS,NODE` solution command and the `PLOT,STRE` command require a projection of element variables to nodes. A continuous stress field is assumed to obtain the nodal values. Accordingly,

$$\sigma = N_{\alpha} \sigma_{\alpha}$$

where σ 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. The projection routine uses a diagonal weight matrix to project the values. For simple elements the matrix is computed by a procedure identical to mass lumping. For example,

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

defines a ‘row sum’ form of projection^[3, 4]. Using the above results in the set of equations and a least square fit with the finite element values $\hat{\sigma}$ gives the equation set

$$M_{\alpha\alpha} \sigma_\alpha = \int_{\Omega} N_\alpha \hat{\sigma} \, d\Omega$$

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 actual solution is performed automatically by *FEAP*.

To permit each element to project its own quantities it is necessary to add the projection operations for each element under `ISW = 8`. These are performed locally for each element similar to all other operations. Figure 5.9 shows a simple 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 `ST(nen,*)` array so as to get correct results. Also, when results are displayed it is necessary to plot results by material type to obtain correct indications of stress discontinuities at material interfaces.

5.6 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 pointer value of `NH1` for the double precision array `HR` in the blank common; similarly data for t_{n+1} starts at pointer value of `NH2`, and that not associated with a time at `NH3`. The three pointers are passed to each element routine in the labeled common

```

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

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

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

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

      integer i,l,lint,nel,nes
      real*8  xg,p(*),s(nen,*),xsj(*),sig(nes,*),shp(nel,*),sg(3,9)

      do l = 1,lint
        do i = 1,nel
          xg = shp(i,l)*xsj(l)
          p(i) = p(i) + xg
          s(i,1) = s(i,1) + sig(1,l)*xg
          s(i,2) = s(i,2) + sig(2,l)*xg
          s(i,3) = s(i,3) + sig(3,l)*xg
          s(i,4) = s(i,4) + sig(4,l)*xg
        end do ! i
      end do ! l
      iste = 4 ! Returns number projections

      end

```

Figure 5.9: Element variable projection routine

```

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

```

5.6.1 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.

5.6.2 Accessing history data for each element

As noted above the data for each element is contained in arrays whose first word in the blank is located at `hr(nh1)`, `hr(nh2)`, and `hr(nh3)` for the t_n , t_{n+1} , and that not associated with time, respectively (note that there are values for each only if the `nh1` or `nh3` were set during the `isw = 1` task. Any other allocated data follows immediately after each first word. It is a user's 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(nh1)`, or `hr(nh1)` types the new quantities will only be returned to the H history for the element 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
```

5.7 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),iepl(2,20),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

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 5.11: Momenta and Energy Assignments

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

the stored energy as

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

and the work by external loads as

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

The value of the displacement and velocity at the current time t_{n+1} are passed in $\mathbf{u1}(i, j, 1)$ and $\mathbf{u1}(i, j, 4)$, respectively. Note that this is true no matter which time integration algorithm is specified.

5.8 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

$$\epsilon_{ss} = \frac{\partial u_s}{\partial s} + \frac{1}{2} \sum_{j=1}^{d-1} \left(\frac{\partial u_{nj}}{\partial s} \right)^2$$

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

$$\delta\epsilon_{ss} = \frac{1}{L} [\delta\mathbf{u}_1 \quad \delta\mathbf{u}_2] \cdot \begin{bmatrix} -\mathbf{g} \\ \mathbf{g} \end{bmatrix}$$

Substituting the above virtual strain expression into the weak form gives the modified residual expression

$$\mathbf{R}_i = \frac{1}{2} b_i L \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \sigma_{ss} A \begin{bmatrix} -g_i \\ g_i \end{bmatrix} - \rho A \frac{L}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} \ddot{u}_{i1} \\ \ddot{u}_{i2} \end{bmatrix} \quad (5.3)$$

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

$$d\epsilon_{ss} = [\mathbf{g}] \cdot \frac{\partial d\mathbf{u}}{\partial s}$$

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.

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.^[1, 2, 3, 4] 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

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

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

$$\sum_{l=1}^L W_l = 2 \quad (6.2)$$

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

```
CALL INT1D ( L , SG , WG )
```

in which L is assigned an integer value between 1 and 5 returns the points in the array $SG(*)$ and the weights in $WG(*)$, both of which are of type $REAL*8$. 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

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

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 acontaining the points and weights according to: $SW(1,1)$ contains values of the points ξ_l ; $SW(2,1)$ contains values of the points η_l ; and $SW(3,1)$ contains values of the weights W_l .

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

```
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: $SW(1,1)$ returns the area coordinate L_{1l} (as defined in [3, 4]); $SW(2,1)$ returns the area coordinate L_{2l} ; $SW(3,1)$ returns the area coordinate L_{3l} ; $SW(4,1)$ returns the weight W_l ; Table 6.1 describes the admissible types, number and location of quadrature points.

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 ($O(h^4)$)
6	6	Nodal ($O(h^3)$)
7	7	Interior ($O(h^6)$)

Table 6.1: Quadrature for triangles

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

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

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: $SW(1,1)$ contains values of the points ξ_l ; $SW(2,1)$ contains values of the points η_l ; and $SW(3,1)$ contains values of the points ζ_l ; and $SW(4,1)$ 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: $SW(1,1)$ returns the volume coordinate $L_{1,l}$ (as defined in [3, 4]); $SW(2,1)$ returns the volume coordinate $L_{2,l}$; $SW(3,1)$ returns the volume coordinate $L_{3,l}$; $SW(4,1)$ returns the volume coordinate $L_{4,l}$; $SW(5,1)$ returns the weight W_l ; Table 6.2 describes the admissible types, number and location of quadrature points.

Type	Number Points	Location
1	1	Centroid ($O(h^2)$)
2	4	Interior ($O(h^3)$)
3	11	Interior ($O(h^4)$)
4	16	Interior ($O(h^5)$)

Table 6.2: Quadrature for tetrahedra

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

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

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

and quadrature may be used for evaluation.

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

```
CALL SHAP1D ( S , NEL, SHP )
```

where

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

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

$$w = N_1^w \bar{w}_1 + N_2^w \bar{w}_2 + N_1^\theta \bar{\theta}_1 + N_2^\theta \bar{\theta}_2 \quad (6.6)$$

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

6.2.2 Shape functions in two-dimensions

Two-dimensional C_0 isoparametric interpolation on quadrilaterals of linear or quadratic 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	Largest local node number on element
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 or quadratic order may be obtained using the subprogram call

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

where

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 (1,2 or 3)
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 heirarchical form for midside nodes.

6.2.3 Shape functions in three-dimensions

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 )
```

where

Parameter	Description
SS(3)	Natural coordinates ξ, η, ζ
XL(NDM,8)	Element coordinates in local order
NDM	Spatial dimension mesh (2 or 3)
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, 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 (2 or 3)
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 Mesh plot routines

FEAP has plot capabilities to represent some element shapes. By default it is assumed that all plots are for two dimensional elements of quadrilateral shape and with between 4 and 9 nodes (numbered as shown in the manuals - vertex nodes first, midside nodes, center node). If other shapes are to be represented it is necessary to include drawing instructions in the element routine segment where material properties are input (i.e., `ISW = 1`). A parameter describing the type of element and named `IEL` is contained in the `COMMON /ELDATA/` and, thus, this common must be included to activate the plot option call. The known types of plots are:

1. Point element with one node obtain by call

```
CALL PLTPT1 ( IEL )
```

2. Line element with two nodes obtained by call

```
CALL PLTLN2 ( IEL )
```

3. Triangular element with 3-nodes obtained by call

```
CALL PLTRI3 ( IEL )
```

4. Triangular element with 6-nodes obtained by call

```
CALL PLTRI6 ( IEL )
```

5. Quadrilateral element with 4-nodes obtained by call

```
CALL PLQUD4 ( IEL )
```

6. Tetrahedral element with 4-nodes obtained by call

```
CALL PLTET4 ( IEL )
```

7. Brick element with 8-nodes obtained by call

```
CALL PLBRK8 ( IEL )
```

Using these and internal extraction of element surfaces the program is able to make some hidden surface plots in three dimensions. Failure to include a plot sequence will usually result in unpredictable plots of the mesh and contour values.

Bibliography

- [1] M. Abramowitz and I.A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publications, New York, 1965.
- [2] T.J.R. Hughes. *The Finite Element Method: Linear Static and Dynamic Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1987.
- [3] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method*, volume 1. McGraw-Hill, London, 4th edition, 1989.
- [4] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method: The Basis*, volume 1. Butterworth-Heinemann, Oxford, 5th edition, 2000.