# *FEAP - - A Finite Element Analysis Program*

*Version 6.3 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

January 1998

# Contents


**1 Introduction** **1**
- 1.1 Setting Program Capacity and Options . . . . . . . . . . . . . . 1
- 1.2 Uses of Common and Include Statements . . . . . . . . . . . . 4

**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

**5 Adding Elements** **16**
- 5.1 Non-linear Transient Solution Forms . . . . . . . . . . . . . . . 21
- 5.2 Example: 2-Node Truss Element . . . . . . . . . . . . . . . . . . 27
  - 5.2.1 Theory for a Truss . . . . . . . . . . . . . . . . . . . . . . 28
- 5.3 Additional Options in Elements . . . . . . . . . . . . . . . . . . 32
  - 5.3.1 Task 1 Options . . . . . . . . . . . . . . . . . . . . . . . 32
  - 5.3.2 Task 3 Options . . . . . . . . . . . . . . . . . . . . . . . 36
- 5.4 Elements with History Variables . . . . . . . . . . . . . . . . . . 37
  - 5.4.1 Assigning amount of storage for each element . . . . . 38
  - 5.4.2 Accessing history data for each element . . . . . . . . . 38
- 5.5 Energy Computation . . . . . . . . . . . . . . . . . . . . . . . . . 39
- 5.6 A Non-linear Theory for a Truss . . . . . . . . . . . . . . . . . . 41

# 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, `FEAP63`, 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 `FEAP63` 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 FEAP63 module. For example:

```
file(1) = 'c:/Feap6_0/Manual/Mesh/'
file(2) = 'c:/Feap6_0/Manual/Macr/'
file(3) = 'c:/Feap6_0/Manual/Plot/'
```

defines a path for a PC system. *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 FEAP63 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 FEAP63 module by setting the variables DEFALT and PROMPT. Setting DEFALT to true indicates that all default options are to be set automatically. If DEFALT is set false, a prompt for contour intervals may be requested by setting PROMPT to true.

*FEAP* has options to produce encapsulated PostScript output files in either gray scale of in color. The default mode may be established by setting the variable PSCOLR and PSREVS. Setting PSCOLR true indicates the PostScript

files will be in color (unless set otherwise by the plot **COLOr** 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 `INCRED` 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 `FEAP63` 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`.

# 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 during inputs 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 a 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 must be performed.

```
real*4  t
integer j
logical errck

errck = pinput (td, 5)

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

will perform the assignments. 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)
```

## 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 $2 \times 3$ 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 `hr(2)` will use the same memory location as `mr(1)` and `mr(2)`. This mechanism permits references to positions in `hr` beyond 1 and `mr` beyond 100 only 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 `FEAP63` module. For some compilers indexing is different for short and long 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. However, using this scheme permits direct reference to either `real*8` or `integer` arrays in program modules.

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 `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` and an integer array with length `NUMEL` (these parameters are contained in `COMMON /CDATA/` and available using an include file `cdata.h`. The arrays will be 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, ipr )
setvar = palloc( 112, 'TEMP2', numel,   1 )
```

These arrays are then 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:

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

| 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

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, ipr )
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 released. 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.

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 `FEAP63`.

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 single 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)
```

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

      logical   uprt

      include  'umac1.h'  ! Contains UCT variable

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 MANUal,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

      logical   uprt
      character lct*15
```

```
      real*8    ctl(3)

      include  'umac1.h'     ! Contains the variable UCT

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`.

# 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 5.1[1]. Some of the options for additional data passed through common blocks is shown in Figure 5.3. Also, in Figure 5.4 the reference to common blocks using include statements is shown. In the prototype routine the number of nodes on an element (`nen`) which is used to dimension `ul` is passed in the labeled common `/cdata/`. Additional discussion is given below on use of some of the other data passed through the common blocks.

Each element can carry out tasks according the value of the task parameter `isw`. A list of currently available tasks is shown in Table 5.2. It is not necessary that all options be coded. However, to use the features available

[1]Note 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.

```
      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         hr(np      ) projection weight: wt(numnp)
c         hr(np+numnp) projection values: vl(numnp,8)
c                       (default: project 8 quantities)
      endif
      end
```

Figure 5.2: *FEAP* Element Subprogram. Part 2

| 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(ndf,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

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

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

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

integer         ior,iow
common /iofile/ ior,iow

integer         nph
real*8                  erav
common /prstrs/ nph,ner,erav

real*8  hr
common  hr(1000)
```

Figure 5.3: *FEAP* Element Common Blocks

```
include  'bdata.h'
include  'cdata.h'
include  'eldata.h'
include  'eltran.h'
include  'hdata.h'
include  'iofile.h'
include  'prstrs.h'
include  'blkcom.h'
```

Figure 5.4: *FEAP* Element Common Blocks using Includes

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.x and tasks 12 and 14 may be necessary. Finally, if special plotting options are desired 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 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:

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

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

| Task | Description | Access Command |
|---|---|---|
| 0 | Output label | `SHOW,ELEM` |
| 1 | Input `d(*)` parameters | Mesh:`MATE,n` |
| 2 | Check elements | Soln:`CHECk` |
| 3 | Compute tangent/residual | Soln:`TANG` |
| | Store in `S`/`r` | `UTAN` |
| 4 | Output element variables | Soln:`STRE` |
| 5 | Compute cons/lump mass | Soln:`MASS` |
| | Store in `S`/`r` | `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 | – | – |
| 17 | Set after activation | Soln:`ACTI` |
| 18 | Set after deactivation | Soln:`DEAC` |
| 19 | – | – |
| 20 | Element plotting | Plot:`PELM` |

Table 5.2: Task Options for *FEAP* Element Subprogram

step algorithm with each quantity defined only at discrete times $t_n$. Accordingly, we have displacements $\mathbf{u}_i(t_n)$ with velocities and accelerations denoted as

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

and

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

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

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

and

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

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

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

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

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

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

and solve

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

The solution is updated using

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

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

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

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

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

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

and

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

define the relationships between the increments. Note that only scalar multipliers involving $\beta$, $\gamma$, and $\Delta t$ are involved between the different measures.

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

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

Note that from the Newmark formulas

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

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

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

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

displacements is

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

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

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

where the $c_i$ are scalar quantities involving the integration parameters of the method selected and $\Delta t$. 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 `s(nst,nst)`, where `nst` is the product of `ndf` and `nen`, with `ndf` the *maximum number of degree-of-freedoms at any node in the problem* and `nen` the maximum number of nodes on any element. The ordering of the unknowns into `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 node attached to the element. Thus

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

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

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

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

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

$$\mathbf{R}_i = \begin{bmatrix} R^i_1 \\ R^i_2 \\ R^i_3 \\ \vdots \\ R^i_{ndf} \end{bmatrix}$$

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

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

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

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

The arrays `xl(i,j)`, `ul(i,j,1)`, `ul(i,j,4)` and `ul(i,j,5)` (described in Table 5.1) are used to obtain the nodal coordinates, displacements,velocities and accelerations, respectively. When *FEAP* solves a problem without transient loading (e.g., inertial loading as mass times acceleration) the velocities and accelerations are set to zero prior to calling the element subroutine. Consequently, in programming the steps to compute the residual `r` the inertia terms have no effect for static or quasi-static problems and may be included (generally there are very few additional operations involved to add these terms). The programming of the tangent array, however, must distinguish between cases in which transient (e.g., inertial) loads are present and those in which they are omitted. The different cases are implemented in *FEAP* by making appropriate assignments to the $c_i$ parameters. To facilitate the programming of the tangent array returned in `s` for the various cases, a parameter array `ctan(3)` is passed to the subprogram in labeled common

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

Table 5.3: Tangent Parameters

`eltran`. When the task parameter `isw` is 3, the values in the `ctan` array are interpreted according to Table 5.3.

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

## 5.2 Example: 2-Node Truss Element

An element routine carries out tasks according to the value assigned to the parameter `isw` as indicated in Table 5.2 To describe basic steps to program the various tasks defined by `isw`, we consider next the problem of a 2-node, linear elastic truss element for small deformation applications. The element is described in sufficient generality to permit solution of both two and three dimensional truss problems.

### 5.2.1 Theory for a Truss

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

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

the strain-displacement equation for small deformations:

$$\epsilon_{ss} = \frac{\partial u_s}{\partial s}$$

and a constitutive equation. For example, considering a linear elastic material the constitutive equation may be written as

$$\sigma_{ss} = E\, \epsilon_{ss}$$

Boundary and initial conditions must also be specified to obtain a well posed problem; however, our emphasis here is the derivation of the element arrays associated with the above differential equations. In the above:

- $s$ is the coordinate along the truss member axis,
- $b_s$ is a loading in direction $s$ per unit length,
- $A$ is the truss cross-section area,
- $\rho$ is the mass density per unit volume,
- $u_s$ is a displacement in direction $s$,
- $\dot{v}_s$ is an acceleration in direction $s$ $(v = \dot{u})$,
- $\epsilon_{ss}$ is a strain along the truss member axis, and
- $\sigma_{ss}$ is the stress on a truss cross section.

The equations may also be deduced from the variational equation

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

$\delta\Pi_{ext}$ contains the boundary and loading terms not associated with an element. Where, in addition to previously defined quantities, we define:

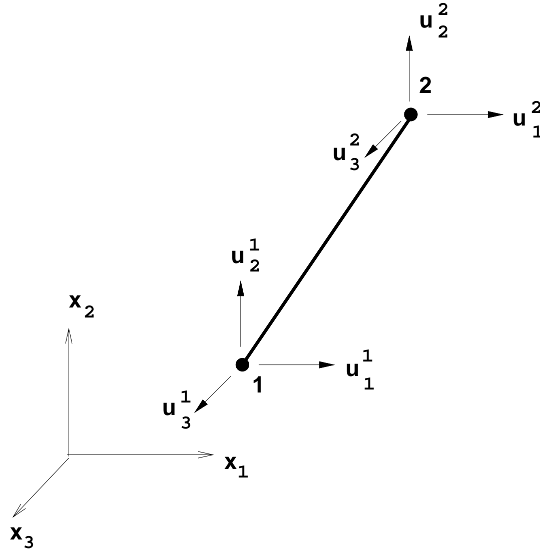


Figure 5.5: 2-Node Truss Element

- $d$ 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,
- $\delta u_i$ 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 unix 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

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

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 $\delta u_i$, etc. The truss strain is

$$\epsilon_s s = \frac{\partial u_s}{\partial s} = \sum_{i=1}^{d} l_i \, \frac{\partial u_i}{\partial s}$$

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

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

$$\left. - \int_L \begin{bmatrix} 1 - \xi \\ \xi \end{bmatrix} b_i ds \right\} + \delta \Pi_{ext}$$

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

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

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

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 $\sigma_{ss}$ 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

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

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 5.1. 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 `r(ndf,nen)` is shown in Figure 5.6.

Similarly, using the results from Section 5.1, the tangent matrix for the truss may be programmed as indicated in Figures 5.7 and 5.8.

## 5.3 Additional Options in Elements

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

### 5.3.1 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 5.2, 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.

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

c       Compute element length

        L2= 0.0d0
        do i = 1,ndm
          L2 = L2 + (xl(i,2) - xl(i,1))**2
        end do
        L = sqrt(L2)

c       Compute strain-displacement matrix

        Lr  = 1.d0/L2
        eps = 0.0d0
        do i = 1,ndm
          bb(i,1) = -(xl(i,2) - xl(i,1))*Lr
          bb(i,2) = -bb(i,1)
          eps     =  eps + bb(i,2)*(ul(i,2,1) - ul(i,1,1))
        end do

c       Compute mass terms

        cmd = rhoA*L/3.0d0
        cmo = cmd*0.5d0

c       Form body/inertia force vector (dm = prop. ld.)

        sigA = EA*eps*L
        body = 0.5d0*L*dm
        do i = 1,ndm
          r(i,1) = body*d(6+i)   - bb(i,1)*sigA
     &            - cmd*ul(i,1,5) - cmo*ul(i,2,5)
          r(i,2) = body*d(6+i)   - bb(i,2)*sigA
     &            - cmo*ul(i,1,5) -  cmd*ul(i,2,5)
        end do
```

Figure 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 + (xl(i,2) - xl(i,1))**2
        end do
        L = sqrt(L2)

c       Form stiffness multiplier

        dd = ctan(1)*EA*L

c       Compute strain-displacement matrix

        Lr = 1.d0/L2
        do i = 1,ndm
          bb(i,1) = -(xl(i,2) - xl(i,1))*Lr
          bb(i,2) = -bb(i,1)
          db(i,1) =  dd*bb(i,1)
          db(i,2) = -db(i,1)
        end do
```

Figure 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
        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

| 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 5.4: Element Plot Definition Subprograms

```
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 5.4 and each routine is called as

```
call plname (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 5.4.

### 5.3.2 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

```
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
```

## 5.4 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

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

### 5.4.1 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.

### 5.4.2 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 users responsibility to manage what is retained in each variable type; however, the order of placing the $t_n$ and $t_{n+1}$ data into the `NH1` and `NH2` arrays should be identical. There are no provisions to store integer history variables separately from double precision quantities. It is necessary to cast the integer data as double precision and move to the history location. For example, using the statement

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

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

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

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

Although users may define new values for any of the `hr(nh1)`, `hr(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
```

## 5.5 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
```

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

Table 5.5: Momenta and Energy Assignments

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

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

the angular momenta as:

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

the kinetic energy

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

the stored energy as

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

and the work by external loads as

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

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

## 5.6 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,

$$\mathbf{n}_1 \; = \; \frac{\mathbf{v} \times \mathbf{l}}{\mid \mathbf{v} \times \mathbf{l} \mid}$$

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

$$\delta\epsilon_{ss} \;=\; \frac{\partial \delta \mathbf{u}}{\partial s} \cdot \left[\; \mathbf{g} \;\right]$$

where

$$\mathbf{g} \;=\; \mathbf{1} \;+\; \sum_{j=1}^{d-1} \frac{\partial u_{nj}}{\partial s} \mathbf{n}_j$$

After differentiation of the displacement field the discrete form of the virtual strain is given by

$$\delta\epsilon_{ss} \;=\; \frac{1}{L} \begin{bmatrix} \delta \mathbf{u}_1 & \delta \mathbf{u}_2 \end{bmatrix} \cdot \begin{bmatrix} -\,\mathbf{g} \\ \mathbf{g} \end{bmatrix}$$

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

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

The tangent tensor is obtained by linearizing the residual as shown previously. The only part which is different is the term with $\sigma_{ss}$. Noting that

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

and

$$d\,\delta\epsilon_{ss} = \frac{\partial \delta \mathbf{u}}{\partial s} \cdot \left( \mathbf{n}_1 \otimes \mathbf{n}_1 \;+\; \mathbf{n}_2 \otimes \mathbf{n}_2 \right) \cdot \frac{\partial d\mathbf{u}}{\partial s}$$

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 $\epsilon_{ss}$ 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.