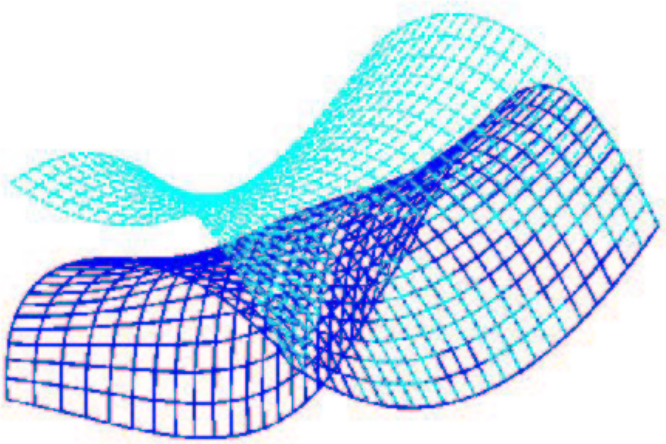
OpenFEM

A finite element toolbox for Matlab

and an obsolete version of Scilab

Revised June 3, 2008



Disclaimer : this manual has few contributors. The maintained parts are contributed by SDTools as parts of the SDT manual

`http://www.sdtools.com/help`

Contents

1

Introduction

OpenFEM

OpenFEM is an open-source software freely distributed under the terms of the GNU Lesser Public License (LGPL).

It is also a registered trademark of INRIA and SDTools, and the corresponding trademark license (under which the name "OpenFEM" may be used) can be found at www-rocq.inria.fr/OpenFEM/trademark.html.

OpenFEM is a finite element toolbox designed to be used within a matrix computing environment. It is available for MATLAB and an unmaintained revision for SCILAB.

OpenFEM is jointly developed by INRIA and SDTools, based on the existing software packages MODULEF and SDT (Structural Dynamics Toolbox). External contributions are strongly encouraged for the forthcoming versions in order to enlarge and improve the toolbox.

1.1 Contact information

`http://www-rocq.inria.fr/OpenFEM`	Web
`openfem@inria.fr`	Mail
`gforge.inria.fr/projects/openfem/`	Project server (source, forum, ...)

1.2 Typesetting conventions and scientific notations

The following typesetting conventions are used in this manual

`courier`	commands, function names, variables
Italics	MATLAB Toolbox names, mathematical notations, and new terms when they are defined
Bold	key names, menu names and items
Small print	comments

Conventions used to specify string commands used by user interface functions are detailed under `commode`.

The following conventions are used to indicate elements of a matrix

`(1,2)`	the element of indices `1`, `2` of a matrix
`(1,:)`	the first row of a matrix
`(1,3: )`	elements `3` to whatever is consistent of the first row of a matrix

Usual abbreviations are

DOF,DOFs	degree(s) of freedom (see section 5.5)
FE	finite element

For mathematical notations, an effort was made to comply with the notations of the

International Modal Analysis Conference (IMAC) which can be found in Ref. [1]. In particular one has

$[\ ],\{\ \}$	matrix, vector
$\bar{}$	conjugate
$[b]$	input shape matrix for model with N DOFs and NA inputs. $\left\{\phi_j^T b\right\}, \left\{\psi_j^T b\right\}$ modal input matrix of the j^{th} normal / complex mode
$[c]$	sensor output shape matrix, model with N DOFs and NS outputs. $\{c\phi_j\}, \{c\psi_j\}$ modal output matrix of the j^{th} normal / complex mode
M, C, K	mass, damping and stiffness matrices
N, NM	numbers of degrees of freedom, modes
NS, NA	numbers of sensors, actuators
$\{q\}_{N\times 1}$	degree of freedom of a finite element model
s	Laplace variable ($s = i\omega$ for the Fourier transform)
$\{u(s)\}_{NA\times 1}$	inputs (coefficients describing the time/frequency content of applied forces)
$[Z(s)]$	dynamic stiffness matrix (equal to $[Ms^2 + Cs + K]$)
λ_j	complex pole
$[\phi]_{N\times NM}$	real or normal modes of the undamped system($NM \leq N$)
$\left[\backslash\Omega^2\backslash\right]$	modal stiffness (diagonal matrix of modal frequencies squared) matrices

1.3 Structural Dynamics Toolbox

Sorry ! Since the OpenFEM manual was originally derived from the *SDT* manual, some links may not yet have been expurged or correspond to functions in the `openfem/sdt3` directory which only have text help (use the `help` command).

SDTools distributes OpenFEM as part of SDT. You can thus an up to date documentation of OpenFEM within the SDT documentation at `http://www.sdtools.com/help`. This documentation being commercial is better maintained that this one.

1.4 Release notes 2008

Sorry for not publishing a post in a long time. Please download source code from the project server `gforge.inria.fr/projects/openfem/`

1.5 Release notes 2006a

OpenFEM has undergone major revisions to get it ready for fully non linear and multi-physics applications. Although these are not fully stabilized a number of key

capabilities are included in this distribution.

- To ease the use for multi-physics problems, DOFs used by an element are now normally dependent on the declared element properties. Standard shapes (`hexa8`, ...) are thus topology holders (8 node volumes) rather than associated with a particular physics formulation. The implementation of a particular set of physics is now entirely defined in the associated property function `p_solid` for 2D and 3D mechanics, linear acoustics and fluid structure coupling, `p_heat` for the heat equation. Other applications not included in the distribution are the generation of layered shell models with variable numbers of layers or the development of poroelastic formulations based in Biot's model. This major change can affect the result of `GetDof` commands when the propertie are not defined.
- Compilation for generic elements has progressed so that you can now define new formulations that include right hand side and stress computations with to need to recompile `of_mk.c` or understand `fe_mknl`. These developments are associated with some performance enhancements and a more consistent set of error reports. `sdtdef('diag',12)` can now be used in a debugging mode for many assembly related problems.
- Non linear 3D solids and follower pressure forces are now supported. This is used in the `RivlinCube` demo that served as starting point of tests of non linear functionalities. Follower pressure is illustrated in `fsc3`.
- The selection of integrations rules in the element properties is now consistently implemented. This is particularly important for non-linear problems but is also used in post-processing applications since it allows stress evaluations at other points than model assembly. OpenFEM can thus be used to post-process stress evaluated in in house codes like GEFDYN.
- Time integration capabilities (`fe_time`) have been significantly enhanced with optimization for explicit integration and implementation of output subsampling techniques that allow for different steps for integration and output. Definitions of time variations of loads is now consistently made using curves (see `fe_curve`).
- an interface to `GMSH` has been introduced to give an access to its interesting unstructured meshing capabilities.

1.5.1 Detail by function

This is an incomplete list giving additional details.

`fsc ...`	compatible fluid structure coupling matrix is now compiled for all 2d topologies and supports geometrically non-linear problems.
`hexa8 ...`	2D and 3D volumes are now topology holders with physics being defined in property functions. Right hand side computations are now supported for generic elements.
`fe_load`	The load assembly was fully revised to optimize the process for non linear operations. Compiled RHS computations for generic elements is now supported.
`p_heat`	solutions to the heat equation problem. This also provides an example of how to extend OpenFEM to the formulation of new problems.
`p_solid`	has undergone a major revision to properly pass arguments to other property functions for constitutive law and integration rule building.
`p_shell`	now supports constitutive law building for classical lamination theory.

1 Introduction

2

Installation

In this chapter, information needed for a correct and a complete installation will be given.

The OpenFEM installation is not the same for Matlab or for Scilab, see the appropriate sections. In section 2.3, advice about a complete and efficient installation are given.

In order to install OpenFEM, you need a C compiler (in the previous versions, a Fortran compiler was needed too, but it is not worth anymore). Precompiled binaries are not provided (neither for the Matlab version nor the for Scilab version).

2.1 Matlab installation

2.1.1 Installation

To install OpenFEM for Matlab you need to

- Download the distribution from the OpenFEM web site.
- Unzip the distribution to the target location of your choice `<installdir>`. Typically `<installdir>=$MATLAB/toolbox`, or if your are a SDT user, choose `<installdir>=$MATLAB/toolbox/sdt`.
  Unzip will create a subdirectory `<installdir>/openfem`.
- For UNIX user, notice that you usually need to become superuser to have write permission in the $MATLAB subdirectories. You can easily circumvent this difficulty by unpacking the distribution in another directory where you have write permission.
- Include OpenFEM in your default path. Open Matlab and run the path check

  ```
  cd(fullfile(matlabroot,'toolbox','openfem'))
  % or cd(fullfile('<installdir>','openfem'))
  ofutil('path')
  ```

  Then save your updated path for future Matlab sessions or include the above lines in your `startup.m` file, see `matlabrc`, (or have your system administrator modify your `$MATLAB/toolbox/local/pathdef.m` file).
- Move the `openfem/html` directory to `$MATLAB/help/toolbox/openfem` if you want it to be seen by MATLAB.
- If you have a source version, you need to compile OpenFEM binaries : enter `ofutil('mexall')` in Matlab window in order to run the compilation step. If you have a binary file version, this step is not needed.

2.1.2 Demos

You will find demos and tests of OpenFEM capabilities in the `openfem/demos` and `openfem/test` directories.

2.2 Scilab installation

Warning : the development is not as active for the Scilab version as for the Matlab one. Please go to `http://www.openfem.net/scilab` for discussion of current status of the Scilab version.

2.2.1 OpenFEM toolbox structure

Figure 2.1 shows the tree directory structure of OpenFEM. The OpenFEM root directory contains the subdirectories `demos`, `doc`, `macros`, `man`, `src` and the files `builder.sce` and `loader.sce`
The `demos` directory contains test files and demonstrations in order to verify the correct running of OpenFEM.
The `doc` directory contains various subdirectories, which contain documentation on OpenFEM (in html and latex).
The `macros` directory contains all OpenFEM functions and a directory named `libop`, which contains functions for compatibility between Scilab and Matlab.
The `man` directory contains OpenFEM man pages.
The `src` directory contains various subdirectories, which contain C and Fortran subroutines and the associated interfaces.

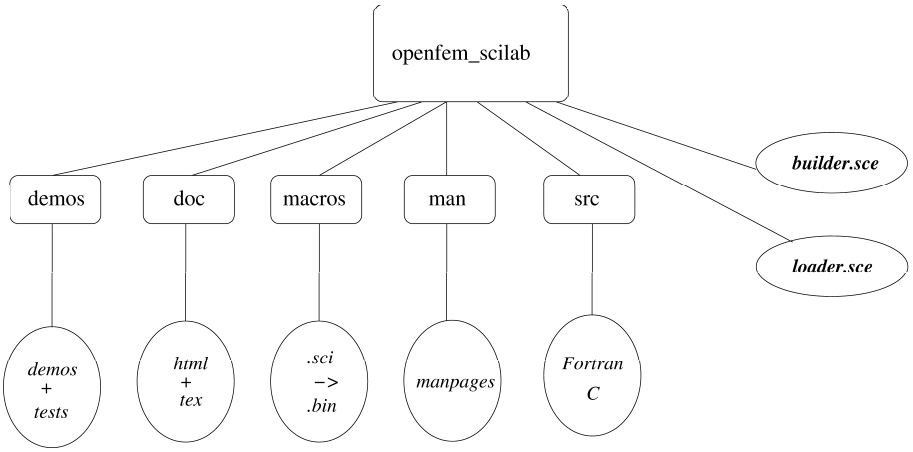


Figure 2.1: OpenFEM for Scilab structure

2.2.2 How to install OpenFEM for Scilab

To install OpenFEM for Scilab you need to :

- Download the distribution from the OpenFEM web site.
- Unzip the distribution to the target location `<installdir>`. Unzip will create a subdirectory `<installdir>/openfem_scilab`.
- If you have a source version, you need ton compile C routines. To compile these routines, go to the `<installdir>/openfem_scilab` subdirectory and run Scilab. Enter `exec builder.sce` in Scilab window : compilation is started. Note that this step is not needed if you have a binary file version.
- To load OpenFEM libraries (once the compilation is successful), enter `exec loader.sce` in the Scilab window. The loading step must be done each time you open a new Scilab window. This procedure can be made automatic : add the following lines to your `.scilab` file.

```
repwd = pwd();
chdir('<installdir>/openfem_scilab'); // specifies access to OpenFEM reposit
exec loader.sce
chdir(repwd);
```

OpenFEM installation is very easy. It can be divided into two steps : compilation of routines and macros, and toolbox loading into Scilab.

For compilation the procedure is the following :

1. move to the OpenFEM directory and run Scilab
2. in Scilab window, enter : `exec builder.sce`

C routines, Fortran routines and macros compilation is now running.

For loading, which is necessary before each use of OpenFEM, the procedure is the following :

1. move to the OpenFEM directory and run Scilab
2. in Scilab window, enter : `exec loader.sce`

Note that this loading can be done automatically. To do so, the user must add the following lines to his/her `.scilab` file or to the `scilab.star` file in Scilab directory.

```
repwd = pwd();
chdir('.../openfem_scilab'); // specify OpenFEM Scilab path
exec ofutil.sce
chdir(repwd);
```

2.2.3 Tests and demonstrations

Tests and demonstrations are provided in the `demos` directory. There are two types of tests : executable files (`.sce`) and function files (`.sci`). Users can see the running of these tests below.

- `.sce` files : move to the `demos` directory and enter `exec filename.sce` in Scilab window. The `.sce` tests are the following :
  - `bar_time.sce` : bar in traction-compression, uses `fe_time`
  - `beambar.sce` : illustrates the use of mixed element types with `femesh`
  - `d_ubeam.sce` : illustrates the use of the `femesh` preprocessor to build a solid model of a U-beam,the computing of the associated modes, and the display of strain energy levels.
  - `gartfe.sce` : illustrates the use of `femesh` to build a small finite element mesh
  - `SL.sce` : tests the `mitc4` element
- `.sci` files : these files are functions. It is necessary to load these functions previously into Scilab. To do so, move to the `demos` directory and enter `getf filename.sci` in Scilab window. When functions are loaded, users can run tests by calling the associated function as follows :
  - `basic_elt_test.sci`
    - `basic_elt_test('integinfo')` : tests basic commands for all elements
    - `basic_elt_test('mat')` : runs elementary tests for all elements
    - `basic_elt_test(st1,st2)` with st1 = 'q4p', 't3p', ... and st2 = 'eig' or 'load' : runs modes computing test ('eig') or loads computing test ('load') for the element specified by st1
    - `basic_elt_test()` : runs `eig` and `load` tests for all elements
  - `test_medit.sci` :
    - `test_medit()` : runs post-processing examples with Medit
    - `test_medit('clean')` : runs post-processing examples with Medit and cleans created files.

2.2.4 Visualization

OpenFEM for Scilab is provided with vizualisation tools. These tools, contained in `feplot` and `fecom` functions, allow the user to see his results quite easily, but they are not very developed. Note that the user can use powerful visualization software through the interface to Medit (`medit.sci`, see OpenFEM documentation, Medit can be downloaded from www-rocq.inria.fr/gamma/medit/). Note also that we encourage users to write interfaces to other visualization packages.
OpenFEM for Scilab visualization is detailed below.

2.2.5 Accepted commands

As stated above, OpenFEM for Scilab visualization is based on the use of `feplot` and `fecom` functions. Accepted commands spelling can be found below.
In the following commands, `node` represents the node matrix, `elt` the model description matrix, `md` the deformations matrix, `dof` the DOFs definition vector.
`model` is a data structure containing at least `.Node` and `.Elt` fields.
`def` is a data structure containing at least `.def` and `.DOF` fields.
`stres` is a vector or a matrix defining stresses in the structure under study.
`opt` is an option vector. opt(1,1) defines the display type : 1 for patch, 2 for lines. opt(1,3) gives the number of deformations per cycle, opt(1,5) the maximum displacement. Other values of `opt` are not used in OpenFEM for Scilab. If the user doesn't want to specify any options, he must replace `opt` by `[]`.
Visualization commands are the following :

- `feplot(node,elt)` : displays the mesh
- `feplot(node,elt,md,dof,opt)` : displays and animates deformations defined by `md`
- `feplot(node,elt,md,dof,opt,stres)` : displays and animates deformations defined by `md` and colors the mesh with `stres` vector.
- `feplot('initmodel',model)` or `feplot('initmodel',node,elt)` : model initialization. The mesh is not displayed. This call is used to prepare the display of deformations by `feplot('initdef',def)`.
- `feplot('initdef',def)` : displays and animates deformations defined by `def.def`. `model` must be previously initialized by a call to another display command or by using `feplot('initmodel',model)`.
- `fecom('colordatastres',stres)` : displays coloring due to `stres` vector. The associated mesh or model must already be known. This call generally follows a deformations display call.

2.2.6 Graphical window description

Most of the commands detailed above open a Scilab graphical window. This window contains specific menus. These menus are detailed below :

- *Display* : display functionalities, patch, color ...
  Contains the following submenus :
  - *DefType* : defines elements display type, with use of wire-frames plots (choose *Line*) or with use of surface plots (choose *Patch*)
  - *Colors* : defines structure coloring. The user can color edges (choose *Lines*), faces (choose *Uniform Patch* if the user decided to represent his

structure with patches). The user can also choose the type of color gradient that he would like to use, if he displays a structure with coloring due to constraints.

- *Parameters* : animation parameters. Used only for deformations visualization. Contains the following subdirectories :
  - *mode +* : for modal deformations, displays next mode.
  - *mode -* : for modal deformations, displays previous mode.
  - *mode number ...* : for modal deformations, allows users to choose the number of the mode to display.
  - *step by step* : allows users to watch animation picture by picture. Press mouse right button to see the next picture, press mouse left button to see the previous picture, press mouse middle button to quit picture by picture animation and to return to continuous animation.
  - *scale* : allows users to modify the displacement scale.
- *Draw* : for non-animated displays. Recovers the structure when it has been erased.
- *Rotate* : opens a window which requests to modify figure view angles. Click on the `ok` button to visualize the new viewpoint and click on the `cancel` button to close the rotation window.
- *Start/Stop* : for deformations animations. Allows users to stop or restart animation.

Remark : To return to Scilab or to continue execution, it is necessary to close the graphical window.

2.2.7 Note on matrix assembly

In OpenFEM for Matlab, renumbering methods are provided with the `fe_mk` function. Users must refer to the OpenFEM documentation for details on the use of these renumbering methods.
In OpenFEM for Scilab, there isn't any renumbering methods provided. So the option `opt(4)` is not referenced. Note that we strongly encourage developers to implement renumbering method for OpenFEM for Scilab.

2.2.8 Note on modal deformations computing

The computation of modal deformations and associated frequencies is done by the `fe_eig` function. For details on accepted commands, the user should refer to OpenFEM documentation. Methods 3, 4, 5, 6 use an eigenvalues computing method based on ARPACK.
ARPACK is a set of Fortran subroutines designed to solve large scale eigenvalues

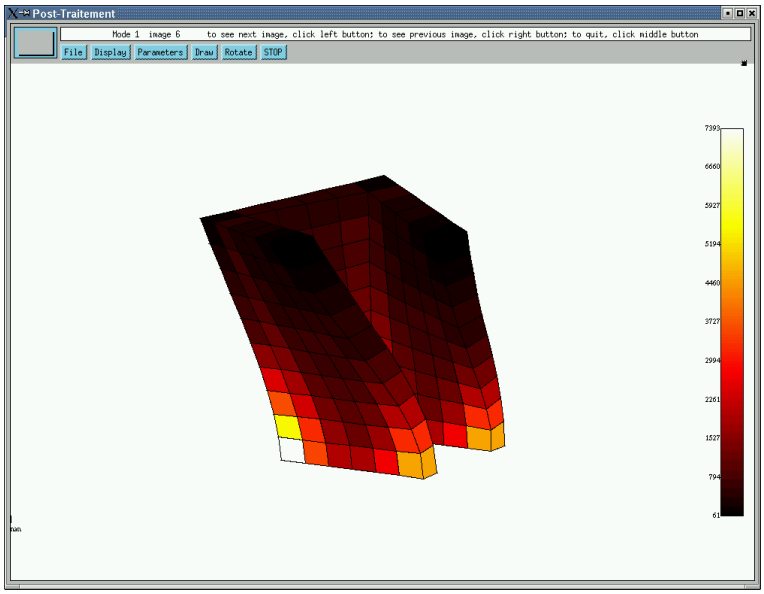


Figure 2.2: OpenFEM for Scilab graphical window

problems (www.caam.rice.edu/software/ARPACK/).
ARPACK is not available under version 2.7 of Scilab. If the user wants to use these resolution methods which are more efficient than the other methods given in `fe_eig`, he must download and install the CVS version of Scilab (www-rocq.inria.fr/Scilab/cvs.html). Note that the use of this method can be faster if the SCISPT toolbox is used. Users can refer to the next section for details on the SCISPT toolbox.

2.2.9 Note on factored matrix object

The factored matrix object (`ofact` function) is designed to let users write code that is independent of the library used to solve static problems of the form $[K]\{q\} = \{F\}$. Users can refer to OpenFEM documentation for more detailed information about `ofact`. In the Matlab version of OpenFEM, a method using UMFPACK is provided.
UMFPACK is a set of routines for solving unsymmetric sparse linear systems. An interface to Matlab is directly integrated in UMFPACK. Users can find more detailed information about UMFPACK at www.cise.ufl.edu/research/sparse/umfpack .
UMFPACK is not directly interfaced with Scilab, so the `fe_eig` method using UMFPACK cannot be directly used. Nevertheless an interface to Scilab is available in the SCISPT toolbox (www-rocq.inria.fr/Scilab/contributions.html). If the user wants to use UMFPACK as a linear solving tool, he can download and install this toolbox in addition to the OpenFEM toolbox and then choose the UMFPACK solver in `ofact`.

2.2.10 Useful information for Matlab users

There are a few differences between Matlab and Scilab. Useful information for users who want to use OpenFEM with both Matlab and Scilab are given here.

- *function call without input* : the call of function without input necessarely is made with empty brackets. For example, in OpenFEM for Scilab, the `fegui` function is be called by `fegui()`, although in OpenFEM for Matlab, it is called by `fegui`.

- *particular functions* : functions like `sum`, `cumsum`, `mean`, `prod`, `cumprod`, used with matrices do not have the same form with one input as in Matlab. Let `a` be a matrix, in Scilab, `sum(a)` gives the sum of all elements of `a`, whereas it gives the sum on the columns of `a` in Matlab.
  In a similar manner, `eye`, `ones` and `zeros` functions are always used with at least two inputs. As a matter of fact, in Matlab `eye(3)` gives an identity matrix of size `3*3`, whereas in Scilab `eye(3)` gives the number 1.

- *nargin, nargout* : `nargin` and `nargout` variables are not automatically defined in Scilab. Users must use the `argn` function as below in order to initialize these variables when they are needed : `[nargout,nargin] = argn()`.

- *data structures compatibility functions* : compatibility functions were implemented in order to optimize the compatibility between OpenFEM for Matlab and OpenFEM for Scilab. In the tests and demonstration provided with OpenFEM, users can observe the use of the `struct` function (defines a data structure in the same manner as Matlab) or the use of the `stack_cell` function (defines a cell array in OpenFEM for Matlab and for Scilab). Note that cell array extraction is lightly different in Scilab. As a matter of fact, Scilab allows extraction only by `(...)` and not by `{...}`.

2.2.11 Note on new element creation

New element display in OpenFEM for Scilab

For information on new element creation, users must refer to the OpenFEM general documentation. If a user wants to add a new element in OpenFEM for Scilab, he must add a standard call with one input in his element file. This call is used to display this new element in OpenFEM for Scilab.

```
if nargin==1 %standard calls with 1 input argument
  if comstr(node,'call')
      idof = ['AssemblyCall']
  elseif ...

  elseif comstr(node,'sci_face')
      idof = ['SciFace']
  ...
```

`SciFace` is a matrix which defines element facets. This matrix must have 3 or 4 columns. This implies that, for elements with less than 3 nodes, the last node must be repeated : `Sciface = [1 2 2]` and that, for elements with faces defined with more than 4 nodes, faces must be cut into subfaces. For example, for a 9-node quadrilateral, `SciFace = [1 5 9 8;5 2 6 9;6 3 7 9;7 4 8 9]`. Orientation conventions are the same as those described in the OpenFEM documentation.

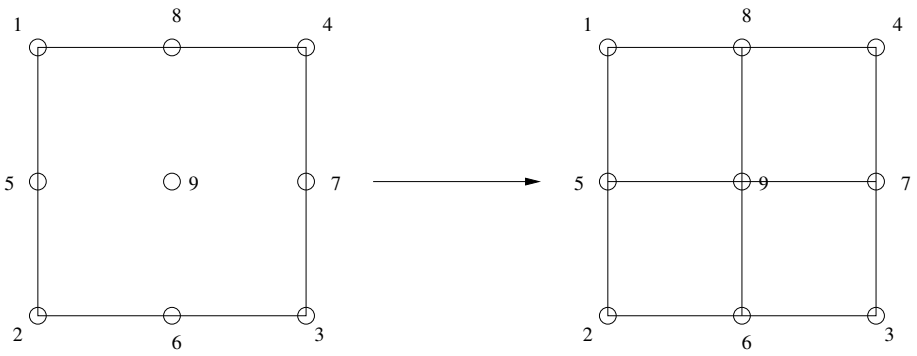


Figure 2.3: Example of cutting in subfaces to display a new element in OpenFEM for Scilab

2.2.12 New element compatibility between OpenFEM for Matlab and OpenFEM for Scilab

It is possible to write a unique code for a new element and to insert it both in OpenFEM for Matlab and in OpenFEM for Scilab. To do so, developers must respect some rules when they are writing a `.m` file. The rules to respect are the following :

- Do not use persistent variables
- Instructions like `k(indice) = fct(...)` or `ks.field = fct(...)` require the use of temporary variables, such as :

  `k(indice)=fct(...)` $\longrightarrow$ `tmp=fct(...); k(indice)=tmp;`
  `ks.field=fct(...)` $\longrightarrow$ `tmp=fct(...); ks.field=tmp;`

- Use the data structures compatibility functions `struct` and `stack_cell` which define respectively a data structure and a cell array. Note that it is impossible to add a new field directly to a data structure. Users must totally define their structures when creating them, or use the following instructions :

  To define a new field `new_field` for the structure `ks` :
  `ks(1)( $ +1) = 'new_field'; ks.new_field = [];`

- Do not use evaluation functions like `eval`, `evalin`, ...
- Do not give the same name to internal functions even if they are contained in different files.

- Always use functions like `sum`, `cumsum`, `mean`, `prod`, `cumprod` with two inputs when the first input is a matrix, which means, for instance, that `sum(a)` when `a` is a matrix must be replaced by `sum(a,1)`.
- Always use functions like `ones`, `zeros`, `eye` with two inputs. For example, `ones(6)` must be replaced by `ones(6,6)`.
- For expressions like `if condition instructions; end` (short form), add a `;` after the condition :

  `if condition instructions; end`

  becomes

  `if conditions; instructions; end`
- Use `size(..,..)` rather than `end` in a matrix :

  `a(end,j)` becomes `a(size(a,1),j)`

  `a(i,end)` becomes `a(i,size(a,2))`

It is very important to respect these rules in order to facilitate the contribution integration in OpenFEM for Scilab. As a matter of fact, if these rules are respected, we can use a translator specific to OpenFEM and then easily integrate the contribution in OpenFEM for Scilab. If a user writes a contribution for OpenFEM for Scilab, we can also integrate it easily in OpenFEM for Matlab. So users can choose in which environement they prefer to contribute, their contributions will be integrated both in OpenFEM for Matlab and OpenFEM for Scilab.

2.2.13 Demos

Demonstrations are provided in the `demos` subdirectory. You can run these demos by moving to the `demos` subdirectory and entering `exec demoname.sce` in Scilab window. If you want to run a demo which is not a script but a function (`.sci` files), you must load the function (`getf demoname.sci`) and then run it (`demoname()` in Scilab window).

2.3 Other useful packages

In order to have a efficient installation of OpenFEM, it is advised to install other packages in addition to OpenFEM.

2.3.1 Medit

Medit is an interactive mesh visualization software. OpenFEM provides an interface to Medit that it is advised to use as an alternative to OpenFEM visualization tools. To use this interface to Medit, you have to install Medit on your computer and to rename the executable as `medit`.

Medit executable is freely available at http://www-rocq.inria.fr/gamma/medit.

You will find details on Medit in the *Tutorial* section.

2.3.2 UMFPACK / SCISPT toolbox

UMFPACK is a set of routines for solving unsymmetric sparse linear systems. It is used in OpenFEM with the factored matrix object (`ofact`, see the section 3.4.4 of the tutorial for details on `ofact`).

An interface to Matlab is directly integrated in UMFPACK. So you just have to install UMFPACK on your computer in order to use it with OpenFEM for Matlab. UMPACK is available at http://www.cise.ufl.edu/research/sparse/umfpack/.

UMFPACK is not directly interfaced with Scilab. Nevertheless an interface to Scilab is available in the SCISPT toolbox (www.scilab.org/contributions.html). So it is advised to install this toolbox in addition to OpenFEM.

2.3.3 Modulef Meshing tool

OpenFEM provides meshing capabilities for simple geometry. In order to handle complex geometries, an interface to Modulef mesh data structure is provided.

`Modulef_Mesh` is a Modulef distribution with only meshers. It is advised to install it if you want to have a complete and powerful finite element computation tool.

`Modulef_Mesh` is available at http://www-rocq.inria.fr/OpenFEM/distrib.html.

2.3.4 GMSH

Another free mesher `http://www.geuz.org/gmsh/` for which an interface into OpenFEM was written.

2.3.5 PARDISO package

The PARDISO package is a very high-performance library designed to solve large sparse symmetric and non-symmetric linear systems of equations (including on shared memory multiprocessors). As for UMFPACK, it can be used in OpenFEM with the factored matrix object `ofact` and the implementation is based on two files: `pardiso_utils.m` in the `@ofact` directory, `pardiso.c` in the `mex` directory.

After obtaining the license key and the compiled library corresponding to your architecture and compiler at http://www.computational.unibas.ch/cs/scicomp/software/pardiso/ for free, you need to compile the `pardiso.c` file and link it to the library by issuing the following command:

```
mex pardiso.c <absolute_path_to_pardiso_lib>/libpardiso_GNU_IA32.so
```

Then move the resulting mex file to the `sdt3` directory. Beware that the resulting executable should be able to find the license key. For more information on where to place the license key file, please refer to the PARDISO documentation. See `ofact`for specific use of PARDISO.

3

Tutorial

NOTE : THIS TUTORIAL HAS NOT BEEN UPDATED IN A LONG TIME IT DOES NOT NECESSARILY REFLECT EXTENSIONS THAT ARE AVAILABLE IN OPENFEM.

This chapter introduces notions needed to use finite element modeling using OpenFEM.
All the examples presented are available for the MATLAB and Scilab versions of OpenFEM. When a difference occurs between these two versions, it is clearly reported.
Furthermore, all scripts mentioned are contained in the `demos` directory of the distribution.

To begin, we explain the typical steps of a finite elements computation below. In a modal analysis case (see the `demo_mode` script) :

- geometry declaration
- handling material and element properties
- defining boundary conditions and constraints
- assembly of mass and stiffness matrices
- normal modes computing
- visualization of deformed structures

In a static analysis case, the steps are almost the same (see the `demo_static` script) :

- geometry declaration
- handling material and element properties
- defining boundary conditions and constraints
- assembly of mass and stiffness matrices
- loads definition
- static response computation
- visualization of deformed structures

The above steps will be explained in the following subsections.

All the scripts listed in this tutorial correspond to the MATLAB syntax. They can be run easily under Scilab with simple modifications : comments (`%`) become `\\`, cell-arrays extraction `{...}` becomes `(...).entries` and functions calls with no input need brackets (for example, a call to the `fegui` function (`fegui` in MATLAB) becomes `fegui()` in Scilab. Note that the cell-arrays definition needs modifications too : `ca = {v1,v2,v3}` becomes `ca = makecell([1 3],v1,v2,v3)` in Scilab.

All the quoted scripts are in the `demos` directory of your OpenFEM installation (either MATLAB or Scilab).

3.1 Declaring finite element models

Before assembly, finite element models are described by a data structure with at least five fields (for a full list of possible fields see section 5.6)

`.Node`	nodes
`.Elt`	elements
`.pl`	material properties
`.il`	element properties
`.Stack`	stack of entries containing additional information cases (boundary conditions, loads, ...), material names, ...

Geometry declarations are described in the sections 3.6.1 (*Direct declaration of geometry*), 3.6.2 (*Geometry declaration with femesh*) and 3.6.3 (*Importing models from other codes*).
Material and element properties handling is presented in the section 3.6.4 and coordinate system handling in section 5.1.2.

Note that, before defining a model, some particular global variables (`FEnode`, `FEel0`, ...) need initializations. This initialization must be done by a call to the `fegui` function: `fegui;` in MATLAB or `fegui();` in Scilab.

3.1.1 Direct declaration of geometry

Hand declaration of a model can only be done for small models and later sections address more complex problems. This example mostly illustrates the form of the model data structure.

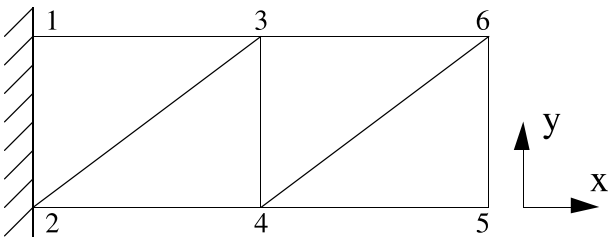


Figure 3.1: FE model.

The geometry is declared in the `model.Node` matrix (see section 5.1). In this case, one defines 6 nodes for the truss and an arbitrary reference node to distinguish principal bending axes (see `beam1`)

```
model = struct('Node',[],'Elt',[]);
%           NodeID  unused   x y z
model.Node=[ 1      0 0 0    0 1 0;
             2      0 0 0    0 0 0;
             3      0 0 0    1 1 0;
             4      0 0 0    1 0 0;
             5      0 0 0    2 0 0;
             6      0 0 0    2 1 0;
             7      0 0 0    1 1 1]; % reference node
```

The model description matrix (see section 5.1) describes 4 longerons, 2 diagonals and 2 battens. These can be declared using three groups of `beam1` elements

```
model.Elt=[
           % declaration of element group for longerons
               Inf     abs('beam1')
           %node1  node2   MatID ProID nodeR, zeros to fill the matrix
               1       3       1    1     7       0
               3       6       1    1     7       0
               2       4       1    1     7       0
               4       5       1    1     7       0
            % declaration of element group for diagonals
               Inf     abs('beam1')
               2       3       1    2     7       0
               4       6       1    2     7       0
            % declaration of element group for battens
               Inf     abs('beam1')
               3       4       1    3     7       0
               5       6       1    3     7       0 ];
```

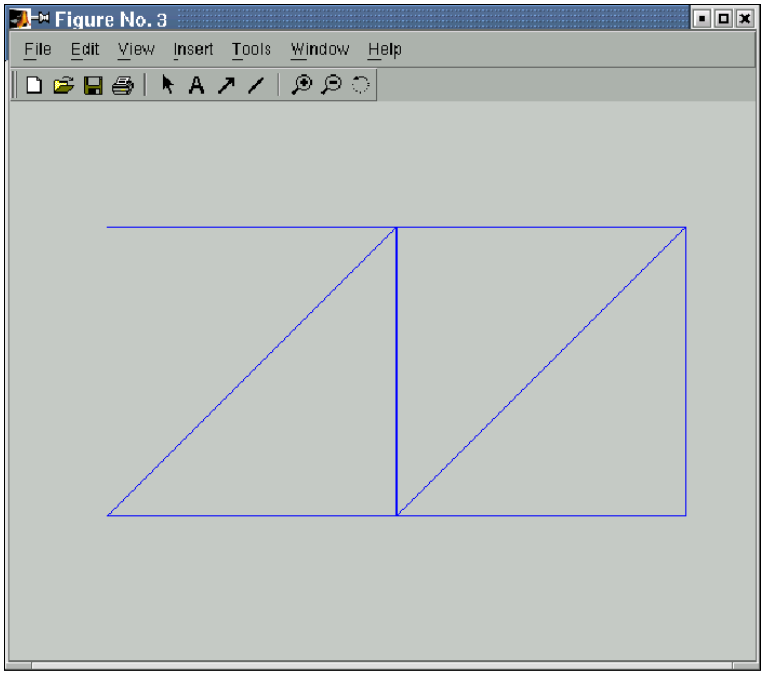
You may view the declared geometry

in Scilab version :

```
feplot(model);
```

in MATLAB version :

```
feplot(model);
fecom('view2');
```



This is the display result in OpenFEM for MATLAB.

The `demo_fe_man` script illustrates uses of this model (part 1, *Direct declaration of geometry*).

3.1.2 Geometry declaration with femesh

Declaration by hand is clearly not the best way to proceed in general. `femesh` provides a number of commands for finite element model creation. The first input argument should be a string containing a single `femesh` command or a string of chained commands starting by a `;` (parsed by `commode` which also provides a `femesh` command mode).

To understand the examples, you should remember that `femesh` uses the following *standard global variables*

`FEnode`	main set of nodes
`FEn0`	selected set of nodes
`FEn1`	alternate set of nodes
`FElt`	main finite element model description matrix
`FEl0`	selected finite element model description matrix
`FEl1`	alternate finite element model description matrix

Two examples are presented below.

- **Fisrt example (`demo_fe` script) :**
  In the example of the previous section, you could use `femesh` as follows: initialize, declare the 4 nodes of a single bay by hand, declare the beams of this bay using the `objectbeamline` command

```
FEnode=[1 0 0 0  0 0 0;2 0 0 0    0 1 0;
        3 0 0 0  1 0 0;4 0 0 0    1 1 0];
femesh('objectbeamline 1 3 0 2 4 0 3 4 0 1 4')
```

  The model of the first bay in is now *selected* (stored in `FEl0`). You can now put it in the main model, translate the selection by 1 in the x direction and add the new selection to the main model

```
femesh(';addsel;transsel 1 0 0;addsel;info');
% export FEnode and FElt geometry in model
model=femesh('model');
feplot(model);
```

  and in Malab version :

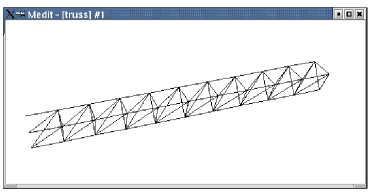
```
fecom('view2');
```

  See the `demo_fe` script, part 1 *Geometry declaration with femesh.*

- **Second example (`d_truss` script) :**
  You could also build more complex examples. For example, one could remove the second bay, make the diagonals a second group of `bar1` elements, repeat the cell 10 times, rotate the planar truss thus obtained twice to create a 3-D triangular section truss and show the result :

```
femesh('reset');
femesh('test2bay')
femesh('removeelt group2');
femesh('divide group 1 InNode 1 4')
femesh('set group1 name bar1');
femesh(';selgroup2 1;repeatsel 10 1 0 0;addsel');
femesh(';rotatesel 1 60 1 0 0;addsel;')
femesh(';selgroup3:4;rotatesel 2 -60 1 0 0;addsel;')
femesh(';selgroup3:8');
% export FEnode and FEl0 geometry in model
model=femesh('model0');
medit('write d_truss',model);
```

See the `d_truss` script, part 1 *Geometry declaration with femesh*.



Visualization of d_truss example with Medit.

`femesh` allows many other manipulations (translation, rotation, symmetry, extrusion, generation by revolution, refinement by division of elements, selection of groups, nodes, elements, edges, etc.) which are detailed in the *Reference* section.

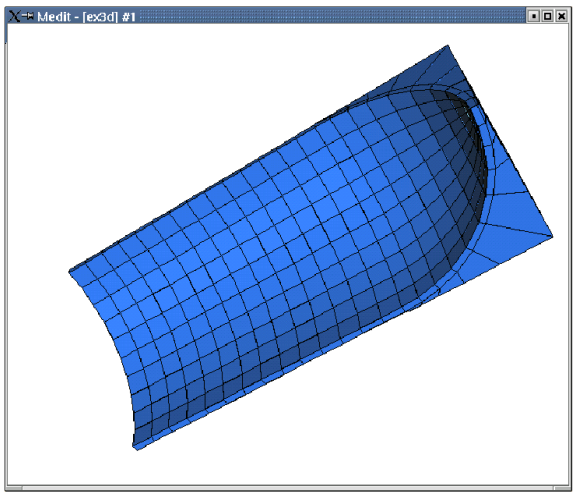
3.1.3 Importing models from other codes

As interfacing with even only the major finite element codes is an enormous and never ending task, such interfaces are always driven by user demands (and supplies !). In this version the interface distributed with OpenFEM is

`nopo` This OpenFEM function reads MODULEF models in binary format.

For example, you can import the model contained in the `ex3d.nopo` file in the `demos` directory (see the `demo_nopo` script).

```
model = nopo('read -p 3d ex3d');
medit('write ex3d',model);
```



Visualization of the `demo_nopo` example with Medit

Other interfaces with major FEM codes are available (for a fee) at www.sdtools.com/tofromfem.html.

3.2 FEM problem formulations

This section gives a short theoretical reminder of supported FEM problems. The selection of the formulation for each element group is done through the material and element properties.

3.2.1 3D elasticity

Elements with a p_solid property entry with a non-zero integration rule are described under p_solid. They correspond exactly to the *b elements, which are now obsolete. These elements support 3D mechanics (DOFs .01 to .03 at each node) with full anisotropy, geometric non-linearity, integration rule selection, ... The elements have standard limitations. In particular they do not (yet)

- have any correction for shear locking found for high aspect ratios
- have any correction for dilatation locking found for nearly incompressible materials

With m_elastic subtypes 1 and 3, p_solid deals with 3D mechanics with strain defined by

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \end{Bmatrix} \begin{bmatrix} N,x & 0 & 0 \\ 0 & N,y & 0 \\ 0 & 0 & N,z \\ 0 & N,z & N,y \\ N,z & 0 & N,x \\ N,y & N,x & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{3.1}$$

and stress by

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sigma_{yz} \\ \sigma_{zx} \\ \sigma_{xy} \end{Bmatrix} = \begin{bmatrix} d_{1,1}N,x+d_{1,5}N,z+d_{1,6}N,y & d_{1,2}N,y+d_{1,4}N,z+d_{1,6}N,x & d_{1,3}N,z+d_{1,4}N,y+d_{1,5}N,x \\ d_{2,1}N,x+d_{2,5}N,z+d_{2,6}N,y & d_{2,2}N,y+d_{2,4}N,z+d_{2,6}N,x & d_{2,3}N,z+d_{2,4}N,y+d_{2,5}N,x \\ d_{3,1}N,x+d_{3,5}N,z+d_{3,6}N,y & d_{3,2}N,y+d_{3,4}N,z+d_{3,6}N,x & d_{3,3}N,z+d_{3,4}N,y+d_{3,5}N,x \\ d_{4,1}N,x+d_{4,5}N,z+d_{4,6}N,y & d_{4,2}N,y+d_{4,4}N,z+d_{4,6}N,x & d_{4,3}N,z+d_{4,4}N,y+d_{4,5}N,x \\ d_{5,1}N,x+d_{5,5}N,z+d_{5,6}N,y & d_{5,2}N,y+d_{5,4}N,z+d_{5,6}N,x & d_{5,3}N,z+d_{5,4}N,y+d_{5,5}N,x \\ d_{6,1}N,x+d_{6,5}N,z+d_{6,6}N,y & d_{6,2}N,y+d_{6,4}N,z+d_{6,6}N,x & d_{6,3}N,z+d_{6,4}N,y+d_{6,5}N,x \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix}$$

Note that the strain states are $\{\epsilon_x \ \ \epsilon_y \ \ \epsilon_z \ \ \gamma_{yz} \ \ \gamma_{zx} \ \ \gamma_{xy}\}$ which may not be the convention of other software. In particular volume elements inherited from MODULEF order shear stresses differently $\sigma_{xy}, \sigma_{yz}, \sigma_{zx}$ (these elements are obtained by setting p_solid integ value to zero. In fe_stres the stress reordering can be accounted for by the definition of the proper TensorTopology matrix.

For isotropic materials

$$D = \begin{bmatrix} \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 \end{bmatrix} & 0 \\ 0 & \begin{bmatrix} G & 0 & 0 \\ 0 & G & 0 \\ 0 & 0 & G \end{bmatrix} \end{bmatrix} \tag{3.2}$$

with at nominal $G = E/\left(2(1+\nu)\right)$.

For constitutive law building, see p_solid.

3.2.2 2D elasticity

With m_elasticsubtype 4, p_solid deals with 2D mechanical volumes with strain defined by (see q4p constants)

$$\left\{ \begin{array}{c} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{array} \right\} = \left[\begin{array}{cc} N,x & 0 \\ 0 & N,y \\ N,y & N,x \end{array} \right] \left\{ \begin{array}{c} u \\ v \end{array} \right\} \tag{3.3}$$

and stress by

$$\left\{ \begin{array}{c} \sigma\epsilon_x \\ \sigma\epsilon_y \\ \sigma\gamma_{xy} \end{array} \right\} = \left[\begin{array}{cc} d_{1,1}N,x + d_{1,3}N,y & d_{1,2}N,y + d_{1,3}N,x \\ d_{2,1}N,x + d_{2,3}N,y & d_{2,2}N,y + d_{2,3}N,x \\ d_{3,1}N,x + d_{3,3}N,y & d_{3,2}N,y + d_{3,3}N,x \end{array} \right] \left\{ \begin{array}{c} u \\ v \end{array} \right\} \tag{3.4}$$

For isotropic plane stress (`p_solid form=1`), one has

$$D = \frac{E}{1-\nu^2} \left[\begin{array}{ccc} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{array} \right] \tag{3.5}$$

For isotropic plane strain (`p_solid form=0`), one has

$$D = \frac{E(1-\nu}{(1+\nu)(1-2\nu)} \left[\begin{array}{ccc} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{array} \right] \tag{3.6}$$

3.2.3 Acoustics

With `m_elastic` subtype 2, `p_solid` deals with 2D and 3D acoustics (see `flui4 constants`) where 3D strain is given by

$$\left\{ \begin{array}{c} p,x \\ p,y \\ p,z \end{array} \right\} = \left[\begin{array}{c} N,x \\ N,y \\ N,z \end{array} \right] \left\{ \begin{array}{c} p \end{array} \right\} \tag{3.7}$$

This replaces the earlier `flui4` ... elements.

3.2.4 Classical lamination theory

3.2.5 Geometric non-linearity

The following gives the theory of large transformation problem implemented in OpenFEM function `of_mk_pre.c Mecha3DInteg`.

The principle of virtual work in non-linear total Lagrangian formulation for an hyperelastic medium is

$$\int_{\Omega_0} (\rho_0 u'', \delta v) + \int_{\Omega_0} S : \delta e = \int_{\Omega_0} f.\delta v \quad \forall \delta v \tag{3.8}$$

with p the vector of initial position, $x = p + u$ the current position, and u the displacement vector. The transformation is characterized by

$$F_{i,j} = I + u_{i,j} = \delta_{ij} + \{N_{,j}\}^T \{q_i\} \tag{3.9}$$

where the N, j is the derivative of the shape functions with respect to cartesian coordinates at the current integration point and q_i corresponds to field i (here translations) and element nodes. The notation is thus realy valid within a single element and corresponds to the actual implementation of the element family in `elem0` and `of_mk`. Note that in these functions, a reindexing vector is use to go from engineering ($\{e_{11}\ e_{22}\ e_{33}\ 2e_{23}\ 2e_{31}\ 2e_{12}\}$) to tensor $[e_{ij}]$ notations `ind_ts_eg=[1 6 5;6 2 4;5 4 3];e_tensor=e_engineering(ind_ts_eg);`. One can also simplify a number of computations using the fact that the contraction of a symmetric and non symmetric tensor is equal to the contraction of the symmetric tensor by the symmetric part of the non symmetric tensor.

One defines the Green-Lagrange strain tensor $e = 1/2(F^T F - I)$ and its variation

$$de_{ij} = \left(F^T dF\right)_{Sym} = \left(F_{ki} \{N_{,j}\}^T \{\delta q_k\}\right)_{Sym} \tag{3.10}$$

Thus the virtual work of internal loads (which corresponds to the residual in non-linear iterations) is given by

$$\int_\Omega S : \delta e = \int_\Omega \{\delta q_k\}^T \{N_{,j}\} F_{ki} S_{ij} \tag{3.11}$$

and the tangent stiffness matrix (its derivative with respect to the current position) can be written as

$$K_G = \int_\Omega S_{ij} u_{k,i} v_{k,j} + \int_\Omega de : \frac{\partial^2 W}{\partial e^2} : \delta e \tag{3.12}$$

which using the notation $u_{i,j} = \{N_{,j}\}^T \{q_i\}$ leads to

$$K_G^e = \int_\Omega \{\delta q_m\} \{N_{,l}\} \left(F_{mk} \frac{\partial^2 W}{\partial e^2}_{ijkl} F_{ni} + S_{lj} \right) \{N_{,j}\} \{dq_n\} \tag{3.13}$$

The term associated with stress at the current point is generally called geometric stiffness or pre-stress contribution.

In isotropic elasticity, the 2nd tensor of Piola-Kirchhoff stress is given by

$$S = D : e(u) = \frac{\partial^2 W}{\partial e^2} : e(u) = \lambda Tr(e) I + 2\mu e \tag{3.14}$$

the building of the constitutive law matrix D is performed in `p_solid BuildConstit` for isotropic, orthotropic and full anisotropic materials. `of_mk_pre.c nonlin_elas` then implements element level computations. For hyperelastic materials $\frac{\partial^2 W}{\partial e^2}$ is not constant and is computed at each integration point as implemented in `hyper.c`.

For a geometric non-linear static computation, a Newton solver will thus iterate with

$$[K(q^n)] \left\{ q^{n+1} - q^n \right\} = R(q^n) = \int_\Omega f.dv - \int_{\Omega_0} S(q^n) : \delta e \tag{3.15}$$

where external forces f are assumed to be non following.

3.2.6 Thermal pre-stress

The following gives the theory of the termoelastic problem implemented in OpenFEM function `of_mk_pre.c nonlin_elas`.

In presence of a temperature difference, the thermal strain is given by $[e_T] = [\alpha]\,(T - T_0)$, where in general the thermal expansion matrix α is proportional to identity (isotropic expansion). The stress is found by computing the contribution of the mechanical deformation

$$S = C : (e - e_T) = \lambda Tr(e)I + 2\mu e - (C : [\alpha])(T - T_0) \tag{3.16}$$

This expression of the stress is then used in the equilibrium (3.8), the tangent matrix computation(3.12), or the Newton iteration (3.15). Note that the fixed contribution $\int_{\Omega_0}(-C : e_T) : \delta e$ can be considered as an internal load of thermal origin.

The modes of the heated structure can be computed with the tangent matrix.

An example of static thermal computation is given in `ofdemos ThermalCube`.

3.2.7 Hyperelasticity

The following gives the theory of the termoelastic problem implemented in OpenFEM function `hyper.c` (called by `of_mk.c MatrixIntegration`).

For hyperelastic media $S = \partial W/\partial e$ with W the hyperelastic energy. `hyper.c` currently supports Mooney-Rivlin materials for which the energy takes one of following forms

$$W = C_1(J_1 - 3) + C_2(J_2 - 3) + K(J_3 - 1)^2, \tag{3.17}$$

$$W = C_1(J_1 - 3) + C_2(J_2 - 3) + K(J_3 - 1) - (C_1 + 2C_2 + K)\ln(J_3), \tag{3.18}$$

where (J_1, J_2, J_3) are the so-called reduced invariants of the Cauchy-Green tensor

$$C = I + 2e, \tag{3.19}$$

linked to the classical invariants (I_1, I_2, I_3) by

$$J_1 = I_1 I_3^{-\frac{1}{3}}, \quad J_2 = I_2 I_3^{-\frac{2}{3}}, \quad J_3 = I_3^{\frac{1}{2}}, \tag{3.20}$$

where one recalls that

$$I_1 = \mathrm{tr}C, \quad I_2 = \frac{1}{2}\left[(\mathrm{tr}C)^2 - \mathrm{tr}C^2\right], \quad I_3 = \det C. \tag{3.21}$$

Note : this definition of energy based on reduced invariants is used to have the hydrostatic pressure given directly by $p = -K(J_3 - 1)$ (K "bulk modulus"), and the third term of W is a penalty on incompressibility.

Hence, computing the corresponding tangent stiffness and residual operators will require the derivatives of the above invariants with respect to e (or C). In an orthonormal basis the first-order derivatives are given by:

$$\frac{\partial I_1}{\partial C_{ij}} = \delta_{ij}, \quad \frac{\partial I_2}{\partial C_{ij}} = I_1\delta_{ij} - C_{ij}, \quad \frac{\partial I_3}{\partial C_{ij}} = I_3 C^{-1}_{ij}, \tag{3.22}$$

where (C^{-1}_{ij}) denotes the coefficients of the inverse matrix of (C_{ij}). For second-order derivatives we have:

$$\frac{\partial^2 I_1}{\partial C_{ij}\partial C_{kl}} = 0, \quad \frac{\partial^2 I_2}{\partial C_{ij}\partial C_{kl}} = -\delta_{ik}\delta_{jl} + \delta_{ij}\delta_{kl}, \quad \frac{\partial^2 I_3}{\partial C_{ij}\partial C_{kl}} = C_{mn}\epsilon_{ikm}\epsilon_{jln}, \tag{3.23}$$

where the ϵ_{ijk} coefficients are defined by

$$\begin{cases} \epsilon_{ijk} & = 0 \quad \text{when 2 indices coincide} \\ & = 1 \quad \text{when } (i,j,k) \text{ even permutation of } (1,2,3) \\ & = -1 \quad \text{when } (i,j,k) \text{ odd permutation of } (1,2,3) \end{cases} \tag{3.24}$$

Note: when the strain components are seen as a column vector ("engineering strains") in the form $(e_{11}, e_{22}, e_{33}, 2e_{23}, 2e_{31}, 2e_{12})'$, the last two terms of (3.23) thus correspond to the following 2 matrices

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/2 \end{pmatrix}, \tag{3.25}$$

$$\begin{pmatrix} 0 & C_{33} & C_{22} & -C_{23} & 0 & 0 \\ C_{33} & 0 & C_{11} & 0 & -C_{13} & 0 \\ C_{22} & C_{11} & 0 & 0 & 0 & -C_{12} \\ -C_{23} & 0 & 0 & -C_{11}/2 & C_{12}/2 & C_{13}/2 \\ 0 & -C_{13} & 0 & C_{12}/2 & -C_{22}/2 & C_{23}/2 \\ 0 & 0 & -C_{12} & C_{13}/2 & C_{23}/2 & -C_{33}/2 \end{pmatrix}. \tag{3.26}$$

We finally use chain-rule differentiation to compute

$$S = \frac{\partial W}{\partial e} = \sum_k \frac{\partial W}{\partial I_k} \frac{\partial I_k}{\partial e}, \tag{3.27}$$

$$\frac{\partial^2 W}{\partial e^2} = \sum_k \frac{\partial W}{\partial I_k} \frac{\partial^2 I_k}{\partial e^2} + \sum_k \sum_l \frac{\partial^2 W}{\partial I_k \partial I_l} \frac{\partial I_k}{\partial e} \frac{\partial I_l}{\partial e}. \tag{3.28}$$

Note that a factor 2 arise each time we differentiate the invariants with respect to e instead of C.

The specification of a material is given by specification of the derivatives of the energy with respect to invariants. The laws are implemented in the `hyper.c EnPassiv` function.

3.2.8 Gyroscopic effects

Written by Arnaud Sternchuss ECP/MSSMat.

In the fixed reference frame which is galilean, the eulerian speed of the particle in $\mathbf{x}$ whose initial position is $\mathbf{p}$ is

$$\frac{\partial \mathbf{x}}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} + \mathbf{\Omega} \wedge (\mathbf{p} + \mathbf{u})$$

and its acceleration is

$$\frac{\partial^2 \mathbf{x}}{\partial t^2} = \frac{\partial^2 \mathbf{u}}{\partial t^2} + \frac{\partial \mathbf{\Omega}}{\partial t} \wedge (\mathbf{p} + \mathbf{u}) + 2\mathbf{\Omega} \wedge \frac{\partial \mathbf{u}}{\partial \mathbf{t}} + \mathbf{\Omega} \wedge \mathbf{\Omega} \wedge (\mathbf{p} + \mathbf{u})$$

$\mathbf{\Omega}$ is the rotation vector of the structure with

$$\mathbf{\Omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

in a (x, y, z) orthonormal frame. The skew-symmetric matrix $[\Omega]$ is defined such that

$$[\Omega] = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

The speed can be rewritten

$$\frac{\partial \mathbf{x}}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} + [\Omega]\,(\mathbf{p} + \mathbf{u})$$

and the acceleration becomes

$$\frac{\partial^2 \mathbf{x}}{\partial t^2} = \frac{\partial^2 \mathbf{u}}{\partial t^2} + \frac{\partial\,[\Omega]}{\partial t}(\mathbf{p} + \mathbf{u}) + 2\,[\Omega]\,\frac{\partial \mathbf{u}}{\partial t} + [\Omega]^2\,(\mathbf{p} + \mathbf{u})$$

In this expression appear

- the acceleration in the rotating frame $\frac{\partial^2 \mathbf{u}}{\partial t^2}$,
- the centrifugal acceleration $\mathbf{a_g} = [\Omega]^2\,(\mathbf{p} + \mathbf{u})$,
- the Coriolis acceleration $\mathbf{a_c} = \frac{\partial [\Omega]}{\partial t}(\mathbf{p} + \mathbf{u}) + 2\,[\Omega]\,\frac{\partial \mathbf{u}}{\partial t}$.

$\mathcal{S}_0^e$ is an element of the mesh of the initial configuration $\mathcal{S}_0$ whose density is ρ_0. $[N]$ is the matrix of shape functions on these elements, one defines the following elementary matrices

$$\begin{array}{rll} \left[D_g^e\right] = & \int_{\mathcal{S}_0^e} 2\rho_0\,[N]^\top\,[\Omega]\,[N]\;d\mathcal{S}_0^e & \textit{gyroscopic coupling} \\ [K_a^e] = & \int_{\mathcal{S}_0^e} \rho_0\,[N]^\top\,\frac{\partial [\Omega]}{\partial t}\,[N]\;d\mathcal{S}_0^e & \textit{centrifugal acceleration} \\ \left[K_g^e\right] = & \int_{\mathcal{S}_0^e} \rho_0\,[N]^\top\,[\Omega]^2\,[N]\;d\mathcal{S}_0^e & \textit{centrifugal softening/stiffening} \end{array} \tag{3.29}$$

3.2.9 Centrifugal follower forces

This is the embrio of the theory for the future implementation of centrifugal follower forces.

$$\delta W_\omega = \int_\Omega \rho\omega^2 R(x)\delta v_R, \tag{3.30}$$

where δv_R designates the radial component (in deformed configuration) of δv. One assumes that the rotation axis is along e_z. Noting $n_R = 1/R\{x_1\ x_2\ 0\}^T$, one then has

$$\delta v_R = n_R \cdot \delta v. \tag{3.31}$$

Thus the non-linear stiffness term is given by

$$-\,d\delta W_\omega = -\int_\Omega \rho\omega^2(dR\delta v_R + Rd\delta v_R). \tag{3.32}$$

One has $dR = n_R \cdot dx (= dx_R)$ and $d\delta v_R = dn_R \cdot \delta v$, with

$$dn_R = -\frac{dR}{R}n_R + \frac{1}{R}\{dx_1\ dx_2\ 0\}^T.$$

Thus, finally

$$-\,d\delta W_\omega = -\int_\Omega \rho\omega^2(du_1\delta v_1 + du_2\delta v_2). \tag{3.33}$$

Which gives

$$du_1\delta v_1 + du_2\delta v_2 = \{\delta q_\alpha\}^T\{N\}\{N\}^T\{dq_\alpha\}, \tag{3.34}$$

with $\alpha = 1, 2$.

3.2.10 Handling material and element properties

Before assembly, one still needs to define material and element properties associated with the various elements.

The properties are stored with one property per row in `pl` and `il` model fields. `model.pl` is a material property matrix and `model.il` is a element property matrix.

A row in the material property matrix begins with a `MatID` which identifies a particular material property and matches with a `MatID` in the model description matrix. Then a `Type` is defined and various material properties are given. See section 5.2 and section 5.3 for details.

A row in the element property matrix as the same shape as a row in the material property matrix. It begins with a `ProID` which is an identifier of a particular element property that matches with a `ProID` in the model description matrix. Then a `Type` is defined and various element property are given. See section 5.4 for details.

When using scripts, it is often more convenient to use low level definitions of the material properties. For example (see the `demo_fe` script, part 2 *Handling material and element properties*) , one can define aluminum and three sets of beam properties with

```
...
%         MatId  MatType                    E       nu    rho
model.pl=[ 1   fe_mat('m_elastic','SI',1)  7.2e+10  0.3   2700 ];
model.il = [ ...
%  ProId SecType                J      I1     I2       A
1 fe_mat('p_beam','SI',1) 5e-9   5e-9   5e-9   2e-5  0 0 % longerons
p_beam('dbval 2','circle 4e-3') % circular section 4 mm
p_beam('dbval 3','rectangle 4e-3 3e-3')%rectangular section 4 x 3 mm
 ];
...
```

To assign a `MatID` or a `ProID` to a group of elements, you can use

- the simple `femesh` set commands. For example `femesh('set group1 mat1 pro3')` will set values 1 to MatID and 3 to ProID for element group 1 (see `gartfe` script).

  An element group is a set of elements of the same type and with the same properties. In a model description matrix (`model.Elt` or `FElt` for example), an element group begins with a header row whose first element is `Inf` and the following the ascii values for the name of the element. It ends by the header row of the next element group or with the end of the model description matrix.

- more elaborate commands based on `femesh` findelt commands. Knowing which column of the `Elt` matrix you want to modify, you can use something of the form (see `gartfe` script)

  `FElt(femesh('find` *`EltSelectors`*`'), `*`IDColumn`*`)=`*`ID`*`;`

You can also get values with `mpid=feutil('mpid',elt)`, modify `mpid`, then set values with `elt=feutil('mpid',elt,mpid)` (see the `demo_fe` script, part 3).

3.2.11 Coordinate system handling

Local coordinate systems are stored in a `model.bas` field described in the `basis` reference section. Columns 2 and 3 of `model.Node` define coordinate system numbers for position and displacement, respectively.

`feplot`, `fe_mk`, `rigid`, ... now support local coordinates. `feutil` does when the model is described by a data structure containing the `.bas` field. `femesh` assumes you are using global coordinate system obtained with

```
[FEnode,bas] = basis(model.Node,model.bas)
```

To write your own scripts using local coordinate systems, it is useful to know the following calls :

`[node,bas,NNode]=feutil('getnodebas',model)` returns the nodes in global coordinate system, the bases `bas` with recursive definitions resolved and the reindexing vector `NNode`.

The command

```
cGL=basis('trans l',model.bas,model.Node,model.DOF)
```

returns the local to global transformation matrix.

3.3 Defining a case

Once the topology (.Node,.Elt, and optionally .bas fields) and properties (.pl,.il fields or associated mat and pro entries in the .Stack field) are defined, you still need to define boundary conditions, constraints (see section 3.7.1) and applied loads before actually computing a response. The associated information is stored in a case data structure. The various cases are then stored in the .Stack field of the model data structure.

Boundary conditions and constraints are detailed in section 3.7.1 and load definitions in section 3.7.2.

3.3.1 Boundary conditions and constraints

Boundary conditions and constraints are described in Case.Stack using FixDof and Rigid case entries (see section 3.7).

FixDof entries are used to easily impose zero displacement on some DOFs. To treat the two bay truss example of section 3.6.1, one will for example use (see demo_fe part 3 *Boundary conditions and constraints*)

```
...
model=fe_case(model,'SetCase1', ...          % defines a new case
 'FixDof','2-D motion',[.03 .04 .05]', ...  % 2-D motion
 'FixDof','Clamp edge',[1 2]');              % clamp edge
...
```

When assembling the model with the specified Case (see section 3.7), these constraints will be used automatically.

Note that, you may obtain a similar result by building the DOF definition vector for your model using a script (**such scripts are considered obsolete since fe_case is more compact and efficient**). Node selection commands allow node selection and fe_c provides additional DOF selection capabilities. In the two bay truss case, (see demo_2bay, part 1)

```
femesh('reset');
model=femesh('test 2bay');
mdof = feutil('getdof group1:2',model);
i1 = femesh('findnode x==0');
adof1 = fe_c(mdof,i1,'dof',1);              % clamp edge
adof2 = fe_c(mdof,[.01 .02 .06]','dof',2); % 2-D motion
adof = [adof1;adof2];
model=fe_case(model,'SetCase1', ...          % defines a new case
    'FixDof','fixed DOF list',adof);
```

finds all DOFs in element groups 1 and 2 of FElt, eliminates DOFs that do not correspond to 2-D motion, finds nodes in the x==0 plane and eliminates the associated DOFs from the initial mdof.

Details on low level handling of fixed boundary conditions and constraints are given in section 5.13.

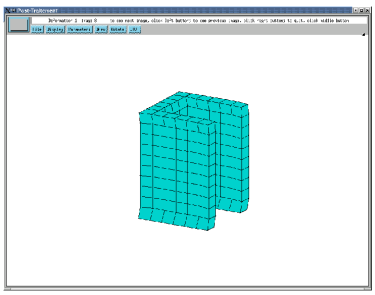
3.3.2 Loads

Loads are described in `Case.Stack` using `DOFLoad`, `FVol` and `FSurf` case entries (see `fe_case`).

Three examples are presented below (see the `demo_ubeam` script).

- To treat a 3D beam example with volume forces (x direction), one will for example use

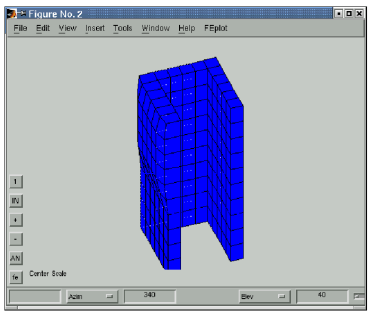
```
femesh('reset');
model = femesh('test ubeam');
data  = struct('sel','GroupAll',...
 'dir',[1 0 0]);                  % defines a force in the x direction
model = fe_case(model,...         % defines a new case
 'FVol','Volume load',data);      % specifies the load type : FVol
Load  = fe_load(model,'case1');   % computes the load
feplot(model,Load);
```



Visualization of volume forces with OpenFEM for Scilab

- To treat a 3D beam example with surface forces, one will for example use

```
femesh('reset');
model = femesh('testubeam');
data=struct('sel','x==-.5', ...      % defines a force : applied on the
   'eltsel','withnode {z>1.25}',...  % elements where x==.5 and z > 1.25
   'def',1,'DOF',.19);
Case1=struct('Stack',stack_cell(...  % defines a case
   stack_cell('Fsurf',...            % specifies the load type : Fsurf
   'Surface load',data)));
Load = fe_load(model,Case1);         % computes the load
feplot(model,Load);
```



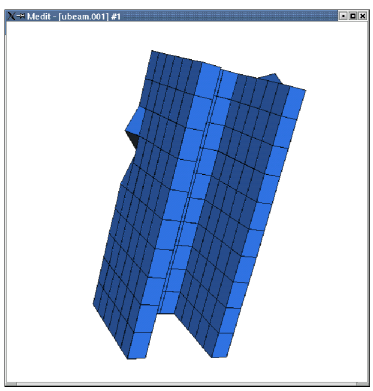
Visualization of surface forces with OpenFEM for MATLAB

- To treat a 3D beam example and create two loads, a relative force between DOFs 207x and 241x and two point loads at DOFs 207z and 365z, one will for example use

```
femesh('reset');
model = femesh('test ubeam');
data  = struct('DOF',...                % defines a force applied on the
 [207.01;241.01;207.03],...             % node 207 (x and z directions)
 'def',[1 0;-1 0;0 1]);                 % and node 241 (x direction)
model = fe_case(model,  ...             % defines case
 'DOFLoad','Point load 1',data);        % specifies the load type : DOFLoad
data  = struct('DOF',365.03,'def',1); % defines a force applied on
                                        % node 365 in the z direction
model = fe_case(model, ...              % defines another case
 'DOFLoad','Point load 2',data);        % specifies the load type : DOFLoad
Load  = fe_load(model,'Case1');         % computes the load
feplot(model,Load);
```

The result of `fe_load` contains 3 columns corresponding to the relative force and the two point loads. You might then combine these forces, by summing them

```
Load.def=sum(Load.def,2);
medit('write visu/ubeam',model,Load,'a',[1 10 0.7]);
```



Visualization of combined forces with Medit

3.4 Computing the response of a model

This section is about the computational part of OpenFEM. Assembly is detailed in section 3.4.1, computing the static response of a structure in section 3.4.2, computing normal modes in section 3.4.3. Moreover, in the section 3.4.4, the case of large finite element models is discussed.

3.4.1 Assembly

Assembly is made by the `fe_mk` function. See the *Reference* section for details on `fe_mk`. Two examples are presented below :

- **First example (`demo_fe` script) :**
  Boundary conditions have already been defined with the use of `fe_case`. These conditions are in the `Stack` field of the `model` structure. In this case, you should use `fe_mk` as follows :

  ```
  model = fe_mk(model);
  ```

  or

  ```
  model = fe_mknl(model);
  ```

  See the `demo_fe` example, part 4 *Assembly*.
  `model` now contains a field `K` which contains mass and stiffness matrices.

- **Second example (`demo_2bay` script) :**
  Boundary conditions are not defined. They can be defined directly in the call of `fe_mk`.

  ```
  femesh('reset');
  model2 = femesh('test 2bay');
  model2 = fe_mk(model2,'FixDof','2-D motion',[.01 .02 .06],...
                 'FixDof','clamp edge',[1 2]);
  ```

  Mass and stiffness matrices are in the `K` field of `model`. They can also be returned in separate matrices :

  ```
  [m,k,mdof] = fe_mk(model2,'FixDof','2-D motion',[.01 .02 .06],...
                     'FixDof','clamp edge',[1 2]);
  ```

  See the `demo_2bay` script, part 2 *Assembly*.

Note that, in OpenFEM for MATLAB (only), `fe_mk` renumbers matrices when the number of DOF is greater than 1000. In this case, the DOF definition vector (`mdof` for example) is modified by the `fe_mk` function. `m` and `k` correpond to the output DOF definition vector.

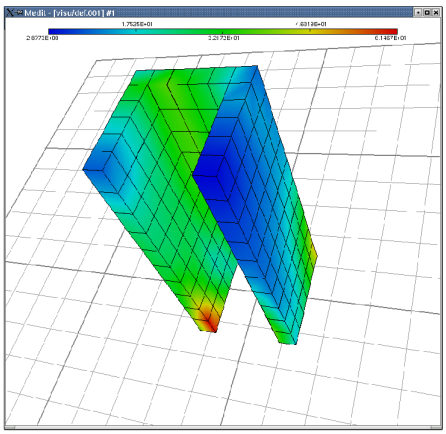
3.4.2 Static response

The computation of the response of static loads can be done as follows.
We suppose that the mass and stiffness matrices have already been assembled (in field `K` of the `model` data structure), and that a load has already been computed in the `Load` data structure.

```
def = struct('def',[],'DOF',model.DOF);
kd = ofact(model.K{2}); % use the factor object for large matrices
def.def = kd\Load.def;
ofact('clear',kd);  % Clear the factor when done
```

You can compute the stress due to the response.

```
Stress = fe_stres('stress mises',model,def);
medit('write visu/def',model,def,Stress,[1 1e8]);
```



Visualization of static response with Medit (`demo_static`)

This example is described in the `demo_static` script.
Note that the animation of the response to static load can be run by clicking with the mouse right button and selecting "Play sequence" in the "Animation" menu.

3.4.3 Normal modes (partial eigenvalues solution)

The computation of normal modes is made by the `fe_eig` function (see the *Function reference* for more details on the use of `fe_eig`). An example of the use of `fe_eig` is shown below.

We suppose that a model has already been defined (in `model` data structure) and that the mass and stiffness matrices have already been assembled (in field `K` of `model`).

```
def=struct('def',[],'DOF',model.DOF,'data',[]);
[def.def,def.data] = fe_eig(model.K{1},model.K{2},[1 4 0 11]);
```

In Scilab, you need to use temporary variables for the `fe_eig` call :

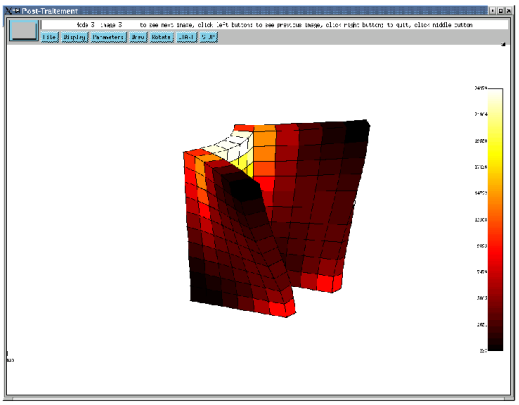
```
def=struct('def',[],'DOF',model.DOF,'data',[]);
[tmpdef,tmpdata] = fe_eig(model.K(1).entries,model.K(2).entries,[1 4 0 11]);
def.def = tmpdef; def.data = tmpdata;
```

Normal modes are in the matrice `def.def` and associated frequencies in the vector `def.data`.
The option vector defines : the method used (in this example : 1), the number of modes to be found (4), the mass shift value needed for rigid body modes (0) and the level of printout (11). For more details, see the *Function reference* section.
The stress due to the deformation can be computed also :

```
StrainEnergy = fe_stres('ener',model,def);
feplot(model.Node,model.Elt,def.def,model.DOF,1,StrainEnergy);
```

and (MATLAB version) :

```
fecom(';color face flat;color edge w;view3');
```



Visualization of a normal mode with OpenFEM for Scilab (`demo_mode`)

This example is described in the `demo_mode` script.

3.4.4 Manipulating large finite element models

This section gives information on manipulating large finite element models.

- Assembly :
  The assembly method can be changed by customizing the `Opt` input (see *Reference functions* section). For large models, it is recommended to use the method 2 (`disk` assembly). The `disk` assembly method uses temporary files and so minimizes memory usage. To change the assembly method, put `opt(3)` to 2.

  You also should allow DOFs with no stiffness to be eliminated (`opt(2)=0`).

  In OpenFEM for MATLAB, an automatic renumbering is done above 1000 DOFs.

  Note that `fe_mknl` is typically much faster than `fe_mk` especially for repeated assembly with the same topology which are usual in non-linear problems.

- Static response :
  The matrix factored object (`ofact`) should be used (see the *Reference functions* section for more details). Furthermore you should install the `umfpack` solver on your machine (`umfpack` is a set of routines for solving unsymmetric sparse linear systems).

  `umfpack` is available at www.cise.ufl.edu/research/sparse/umfpack. An interface to MATLAB is directly integrated in `umfpack`.

  For Scilab, `umfpack` can be used with the help of the `scispt` toolbox (you need to install this toolbox in your machine). This toolbox is available at http://www.scilab.org/contributions.html.

3.5 Visualization of deformed structures

OpenFEM provides various post-processing tools : an OpenFEM specific visualization and an interface to `Medit`.

OpenFEM visualization tools are described in section 3.5.1 and the `Medit` interface in section 3.5.2.

3.5.1 OpenFEM tools

Visualization specific to OpenFEM is implemented in the `feplot` function.

`feplot` supports a number of display types for FE results. The `feplot` provided with OpenFEM (in the `sdt3` directory) is **provided to let you do some post processing with no need to buy a commercial package but clearly is not developed with the same care as the rest of OpenFEM.**

Furthermore, `feplot` is one of the main differences between Scilab and MATLAB versions of OpenFEM. Basic calls to `feplot` are common to both but the use of the graphical window created is very different.

In this section, common calls to `feplot` are described and then specific use of MATLAB and Scilab versions are detailed.

Common calls

As stated above, OpenFEM visualization is based on the use of `feplot` (and `fecom`) function.

In the following commands, `node` represents the node matrix, `elt` the model description matrix, `md` the deformations matrix, `dof` the DOFs definition vector.
`model` is a data structure containing at least `.Node` and `.Elt` fields.
`def` is a data structure containing at least `.def` and `.DOF` fields. `def.def` is a deformation matrix, as `md`.
`stres` is a vector or a matrix defining stresses in the structure under study.

`opt` is an option vector :
`opt(1,1)` defines the display type : 1 for patch, 2 for lines.
`opt(1,2)` defines the Undef type : 00 for none, 01 for UndefDot, 02 for UndefLine (in MATLAB only).
`opt(1,3)` gives the number of deformations per cycle.
`opt(1,4)` defines the number of the node used for modeshape scaling (in MATLAB only).
`opt(1,5)` gives the maximum displacement.
To avoid specifying any option, replace `opt` by `[]`.

Visualization commands are the following :

- `feplot(node,elt)` : displays the mesh

- `feplot(node,elt,md,dof,opt)` : displays and animates deformations defined by `md`
- `feplot(node,elt,md,dof,opt,stres)` : displays and animates deformations defined by `md` and colors the mesh with `stres` vector.
- `feplot('initmodel',model)` or `feplot('initmodel',node,elt)` : model initialization. The mesh is not displayed. This call is used to prepare the display of deformations by `feplot('initdef',def)`.
- `feplot('initdef',def)` : displays and animates deformations defined by `def.def`. `model` must be previously initialized by a call to another display command or by using `feplot('initmodel',model)`.
- `fecom('colordatastres',stres)` : displays coloring due to `stres` vector. The associated mesh or model must already be known. This call generally follows a deformations display call.

For a full list of accepted commands, see the *Reference functions* section or the online help of your OpenFEM version (`help feplot` in MATLAB or Scilab).

MATLAB specific tools

For FE analyses (connectivity specified using a model description matrix `elt`) one will generally use surface plots (type `1` color-coded surface plots using `patch` objects) or wire-frame plots (type `2` using `line` objects). Once the plot is created, it can be manipulated using `fecom`. Continuous animation of experimental deformations is possible although speed is strongly dependent on computer configuration and `figure renderer` selection (use `Feplot:Renderer` menu to switch).

You can initialize plots with

```
feplot(node,elt,mode,mdof,1)
fecom('view3');
```

To get started, run the `d_ubeam` demo. Then

- At this level note how you can zoom by selecting a region of interest with your mouse (double click or press the `i` key to zoom back). You can make the axis active by clicking on it and then use the any of the `u`, `U`, `v`, `V`, `w`, `W`, `2` keys to rotate the plot (look at the `iimouse` help for more possibilities).
- Initialize a set of deformations and show deformation 7 (first flexible mode)

  ```
  feplot(node,elt,mode,mdof,1)
  feplot('initdef',md1,mdof); fecom('ch7');
  feplot('initcdef',StrainEnergy);
  ```
- Scan through the various deformations using the `+/-` buttons/keys. Animate the deformations by clicking on the ▶ button. Notice how you can still change the current deformation, rotate, etc. while running the animation.
- Use `fecom('triax')` to display an orientation triax.
- Use the `fecom('sub 1 2')` command to get a plot with two views of the same mode.

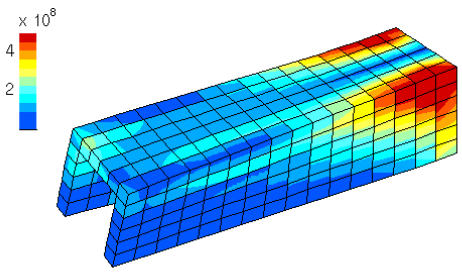


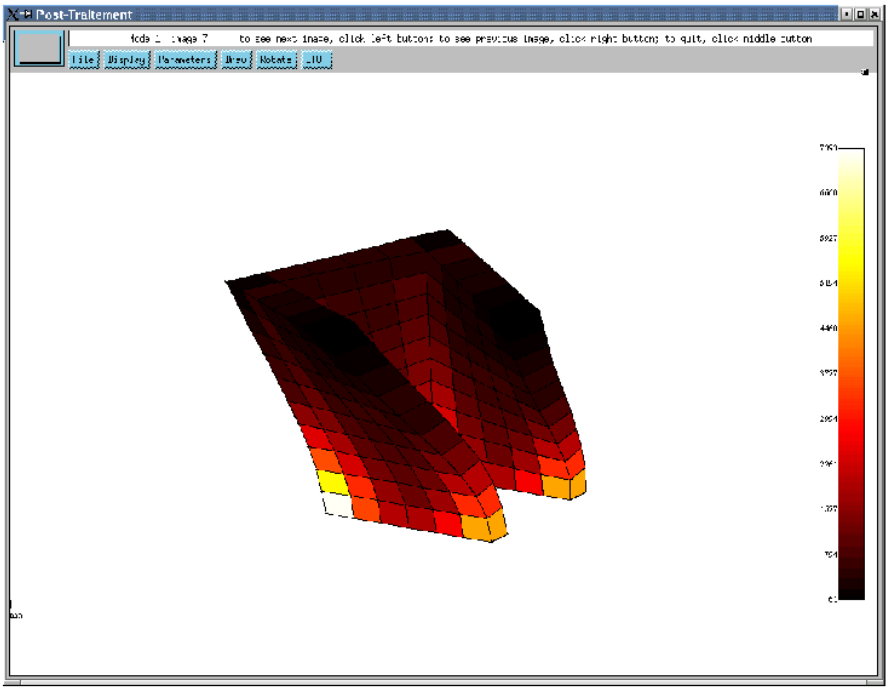
Figure 3.2: Strain energy.

- Note that when you print the figure, you may want to use the `-noui` switch so that the GUI is not printed. Example `print -noui -depsc2 FileName.eps`

Scilab specific tools

Most of the commands detailed in *Common calls* open a Scilab graphical window. This window contains specific menus. These menus are detailed below :

- **Display** : display functionalities, patch, color ...
  Contains the following submenus :
  - **DefType** : defines elements display type, with use of wire-frames plots (choose **Line**) or with use of surface plots (choose **Patch**)
  - **Colors** : defines structure coloring. The user can color edges (choose **Lines**), faces (choose **Uniform Patch** if the user decided to represent the structure with patches). The user can also choose the type of color gradient, if he displays a structure with coloring due to constraints.
- **Parameters** : animation parameters. Used only for deformations visualization. Contains the following subdirectories :
  - **mode +** : for modal deformations, displays next mode.
  - **mode -** : for modal deformations, displays previous mode.
  - **mode number ...** : for modal deformations, allows users to choose the number of the mode to display.
  - **step by step** : allows users to watch animation picture by picture. Press mouse right button to see the next picture, press mouse left button to see the previous picture, press mouse middle button to quit picture by picture animation and to return to continuous animation.
  - **scale** : allows users to modify the displacement scale.
- **Draw** : for non-animated displays. Recovers the structure when it has been erased.
- **Rotate** : opens a window which requests to modify figure view angles. Click on the **ok** button to visualize the new viewpoint and click on the **cancel** button to close the rotation window.
- **Start/Stop** : for deformation animations. Allows users to stop or restart animation.



OpenFEM for Scilab graphical window

Remark : To return to Scilab or to continue execution, it is necessary to close the graphical window.

3.5.2 Visualization with Medit

Visualization with Medit is common to MATLAB and Scilab versions of OpenFEM. Medit is a powerful interactive mesh visualization software, developed by the Gamma project at INRIA-Rocquencourt.
Binaries are freely available at http://www-rocq.inria.fr/gamma/medit.
An interface to Medit is provided. Users need to install Medit themselves if they want to use this interface. They also have to change the name of the Medit executable as medit if it is different.

The interface to Medit is called medit. It allows the same plots and continuous animations as feplot. Details on the use of medit are provided in the section *Function reference* of this document.

To get started, run the test_medit demo : in MATLAB, type 'test_medit' or 'test_medit clean' to clean all the created files. In Scilab, go to the demos directory, load the test ('getf test_medit.sci') and run the test ('test_medit()' or 'test_medit clean').

Then

- note how you can easily move the structure by pressing the left button of the mouse and then moving the mouse. Change the background color by typing 'b'. Now run the animation : press the right button of the mouse, select the '**Animation**' menu and the '**Play sequence**' submenu. Close the Medit window.
- a second window opens. Change the background color by typing 'b'. Display energy constraints : press the right button of the mouse and select the '**Data**' menu and the '**Toggle metric**' submenu. You can run the animation as in previous step. Close the Medit window.
- a third window opens. Change the render mode : press the right button of the mouse, select the '**Render mode**' menu and the '**Wireframe**' submenu. Now choose the submenu '**Shading+lines**' from the menu '**Render mode**' . Display the nodes numbers : press the right button of the mouse, select the '**Items**' menu and the '**Toggle Point num**' submenu. Close the Medit window.
- a final window opens. Change the background color by typing 'b'. Display energy constraints by typing 'm'. Define now a cutting section : press the right button of the mouse, choose '**[F1] Toggle clip**' in menu '**Clipping**'. Press the key function 'F2' : the plane is now selected, you can move it by pressing the left or the middle button of the mouse. Press 'F1' to quit the cutting section environment.

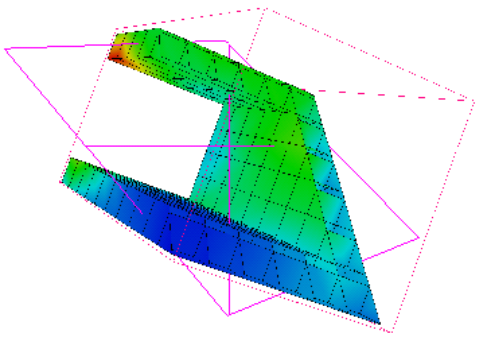


Figure 3.3: Simulation properties tab.

Cutting section with `Medit`

You can find information about the use of `Medit` in the `Medit` documentation (download from the same address as the executable).

3.6 Model data structure

Before assembly, finite element models are described by a data structures with at least five fields (for a full list of possible fields see section 5.6)

`.Node`	nodes
`.Elt`	elements
`.pl`	material properties
`.il`	element properties
`.Stack`	stack of entries containing additional information cases (boundary conditions, loads...), material names...

The following sections illustrate : low level input of nodes and elements in section 3.6.1; structured meshing and mesh manipulation with the `femesh` pre-processor in section 3.6.2; import of FEM models in section 3.6.3. Assembly and response computations are addressed in section 3.7.

3.6.1 Direct declaration of geometry (truss example)

Hand declaration of a model can only be done for small models and later sections address more realistic problems. This example mostly illustrates the form of the model data structure.

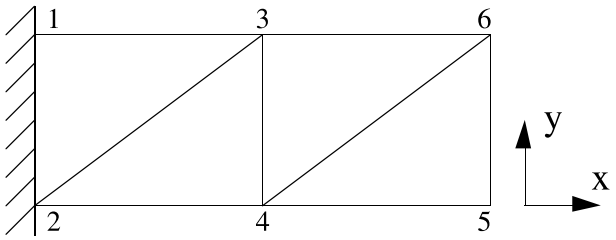


Figure 3.4: FE model.

The geometry is declared in the `model.Node` matrix (see section 5.1). In this case, one defines 6 nodes for the truss and an arbitrary reference node to distinguish principal bending axes (see `beam1`)

```
%     NodeID  unused   x y z
model.Node=[ 1      0 0 0    0 1 0; ...
             2      0 0 0    0 0 0; ...
             3      0 0 0    1 1 0; ...
             4      0 0 0    1 0 0; ...
             5      0 0 0    2 0 0; ...
             6      0 0 0    2 1 0; ...
             7      0 0 0    1 1 1]; % reference node
```

The model description matrix (see section 5.1) describes 4 longerons, 2 diagonals and 2 battens. These can be declared using three groups of `beam1` elements

```
model.Elt=[ ...
      ...     % declaration of element group for longerons
               Inf     abs('beam1') ; ...
      ...     %node1  node2   MatID ProID nodeR, zeros to fill the matrix
               1       3       1    1     7       0 ; ...
               3       6       1    1     7       0 ; ...
               2       4       1    1     7       0 ; ...
               4       5       1    1     7       0 ; ...
      ...      % declaration of element group for diagonals
               Inf     abs('beam1') ; ...
               2       3       1    2     7       0 ; ...
               4       6       1    2     7       0 ; ...
      ...      % declaration of element group for battens
               Inf     abs('beam1') ; ...
               3       4       1    3     7       0 ; ...
               5       6       1    3     7       0 ];
```

You may view the declared geometry

```
feplot(model);
fecom(';view2;textnode;triax;'); % manipulate axes
```

The `demo_fe` script illustrates uses of this model.

3.6.2 Building models with femesh

Declaration by hand is clearly not the best way to proceed in general. `femesh` provides a number of commands for finite element model creation. The first input argument should be a string containing a single `femesh` command or a string of chained commands starting by a `;` (parsed by `commode` which also provides a `femesh` command mode).

To understand the examples, you should remember that `femesh` uses the following *standard global variables*

`FEnode`	main set of nodes
`FEn0`	selected set of nodes
`FEn1`	alternate set of nodes
`FEelt`	main finite element model description matrix
`FEel0`	selected finite element model description matrix
`FEel1`	alternate finite element model description matrix

In the example of the previous section (see also the `d_truss` demo), you could use `femesh` as follows: initialize, declare the 4 nodes of a single bay by hand, declare the beams of this bay using the `objectbeamline` command

```
FEel0=[]; FEelt=[];
FEnode=[1 0 0 0  0 0 0;2 0 0 0    0 1 0; ...
        3 0 0 0  1 0 0;4 0 0 0    1 1 0]; ...
femesh('objectbeamline 1 3 0 2 4 0 3 4 0 1 4');
```

The model of the first bay in is now *selected* (stored in `FEel0`). You can now put it in the main model, translate the selection by 1 in the x direction and add the new selection to the main model

```
femesh(';addsel;transsel 1 0 0;addsel;info');
model=femesh('model');  % export FEnode and FEelt geometry in model
feplot(model);
fecom(';view2;textnode;triax;');
```

You could also build more complex examples. For example, one could remove the second bay, make the diagonals a second group of `bar1` elements, repeat the cell 10 times, rotate the planar truss thus obtained twice to create a 3-D triangular section truss and show the result (see `d_truss`)

```
femesh('reset');
femesh('test2bay');
femesh('removeelt group2');
femesh('divide group 1 InNode 1 4');
femesh('set group1 name bar1');
femesh(';selgroup2 1;repeatsel 10 1 0 0;addsel');
femesh(';rotatesel 1 60 1 0 0;addsel;')
femesh(';selgroup3:4;rotatesel 2 -60 1 0 0;addsel;')
femesh(';selgroup3:8');
model=femesh('model0');  % export FEnode and FEel0 geometry in model
```

```
feplot(model);
fecom(';triaxon;view3;view y+180;view s-10');
```

femesh allows many other manipulations (translation, rotation, symmetry, extrusion, generation by revolution, refinement by division of elements, selection of groups, nodes, elements, edges, etc.) which are detailed in the *Reference* section.

3.6.3 Importing models from other codes

As interfacing with even only the major finite element codes is an enormous and never ending task, such interfaces are always driven by user demands. Interfaces distributed with OpenFEM are

nopo — This OpenFEM function reads MODULEF models in binary format.

You will find an up to date list of interfaces (some paying) with other FEM codes at www.sdtools.com/tofromfem.html).

3.6.4 Handling material and element properties

Before assembly, one still needs to define material and element properties associated to the various elements.

The properties are stored with one property per row in pl (see section 5.3) and il (see section 5.4) model fields.

When using scripts, it is often more convenient to use low level definitions of the material properties. For example , one can define aluminum and three sets of beam properties with

```
femesh('reset');
model=femesh('test 2bay plot');
%         MatId  MatType                    E       nu    rho
model.pl = m_elastic('dbval 1 steel')
model.il = [ ...
... %  ProId SecType                 J      I1     I2       A
1 fe_mat('p_beam','SI',1) 5e-9   5e-9   5e-9   2e-5  0 0 ; ... % longerons
p_beam('dbval 2','circle 4e-3') ; ... % circular section 4 mm
p_beam('dbval 3','rectangle 4e-3 3e-3') ...%rectangular section 4 x 3 mm
 ];
```

To assign a MatID or a ProID to a group of elements, you can use

- the graphical procedure (in the context menu of the material and property tabs, use the Select elements and affect ID procedures and follow the instructions);
- the simple femesh set commands. For example femesh('set group1 mat101 pro103') will set values 101 and 103 for element group 1.

- more elaborate commands based on `femesh` findelt commands. Knowing which column of the `Elt` matrix you want to modify, you can use something of the form (see `gartfe`)

  `FEelt(femesh('find` *`EltSelectors'), IDColumn)=ID*`;`

  You can also get values with `mpid=feutil('mpid',elt)`, modify `mpid`, then set values with `elt=feutil('mpid',elt,mpid)`.

3.6.5 Coordinate system handling

Local coordinate systems are stored in a `model.bas` field described in the `basis` reference section. Columns 2 and 3 of `model.Node` define respectively coordinate system numbers for position and displacement.

`feplot`, `fe_mk`, `rigid`, ... now support local coordinates. `feutil` does when the model is discribed by a data structure with the `.bas` field. `femesh` assumes you are using global coordinate system obtained with

```
[FEnode,bas] = basis(model.Node,model.bas)
```

To write your own scripts using local coordinate systems, it is useful to know the following calls :

`[node,bas,NNode]=feutil('getnodebas',model)` returns the nodes in global coordinate system, the bases `bas` with recursive definitions resolved and the reindexing vector `NNode`.

The command

```
cGL=basis('trans l',model.bas,model.Node,model.DOF)
```

returns the local to global transformation matrix.

3.7 Defining a case

Once the topology (`.Node`,`.Elt`, and optionally `.bas` fields) and properties (`.pl`,`.il` fields or associated `mat` and `pro` entries in the `.Stack` field) are defined, you still need to define boundary conditions, constraints (see section 3.7.1) and applied loads before actually computing a response. The associated information is stored in a case data structure. The various cases are then stored in the `.Stack` field of the model data structure.

3.7.1 Boundary conditions and constraints

Boundary conditions and constraints are described in `Case.Stack` using `FixDof`, `Rigid`, ... case entries (see section 3.7).(`KeepDof` still exists but often leads to misunderstanding)

`FixDof` entries are used to easily impose zero displacement on some DOFs. To treat the two bay truss example of section 3.6.1, one will for example use

```
femesh('reset');
model=femesh('test 2bay plot');
model=fe_case(model, ...          % defines a new case
 'FixDof','2-D motion',[.03 .04 .05]', ...
 'FixDof','Clamp edge',[1 2]');
fecom('promodelinit') % open model GUI
```

When assembling the model with the specified `Case` (see section 3.7), these constraints will be used automatically.

Note that, you may obtain a similar result by building the DOF definition vector for your model using a script. Node selection commands allow node selection and `fe_c` provides additional DOF selection capabilities. Details on low level handling of fixed boundary conditions and constraints are given in section 5.13.

3.7.2 Loads

Loads are described in `Case.Stack` using `DOFLoad`, `FVol` and `FSurf` case entries (see `fe_case` and section 5.7).

To treat a 3D beam example with volume forces (x direction), one will for example use

```
femesh('reset');
model = femesh('test ubeam plot');
data  = struct('sel','GroupAll','dir',[1 0 0]);
model = fe_case(model,'FVol','Volume load',data);
Load  = fe_load(model,'case1');
feplot(model,Load);fecom(';undef;triax;promodelinit');
```

To treat a 3D beam example with surfacic forces, one will for example use

```
femesh('reset');
model = femesh('testubeam plot');
data=struct('sel','x==-.5', ...
   'eltsel','withnode {z>1.25}','def',1,'DOF',.19);
model=fe_case(model,'Fsurf','Surface load',data);
Load = fe_load(model); feplot(model,Load);
```

To treat a 3D beam example and create two loads, a relative force between DOFs 207x and 241x and two point loads at DOFs 207z and 365z, one will for example use

```
femesh('reset');
model = femesh('test ubeam plot');
data  = struct('DOF',[207.01;241.01;207.03],'def',[1 0;-1 0;0 1]);
model = fe_case(model,'DOFLoad','Point load 1',data);
data  = struct('DOF',365.03,'def',1);
model = fe_case(model,'DOFLoad','Point load 2',data);
Load  = fe_load(model);
feplot(model,Load);fecom('textnode365 207 241'); fecom('promodelinit');
```

The result of fe_load contains 3 columns corresponding to the relative force and the two point loads. You might then combine these forces, by summing them

```
Load.def=sum(Load.def,2);
cf.def= Load;
fecom('textnode365 207 241');
```

4

Application examples

This chapter groups theoretical notes associated with OpenFEM demos.

4.1 RivlinCube

Giving us the following displacements on a parallelepiped ($l_1 \times l_2 \times l_3$)

$$u_1 = \lambda_1 x_1, \quad u_2 = \lambda_2 x_2, \quad u_3 = \lambda_3 x_3. \tag{4.1}$$

We obtain for the deformation gradient F and the Cauchy-Green tensor C

$$F = \mathrm{diag}(1+\lambda_i), \quad C = \mathrm{diag}[(1+\lambda_i)^2], \tag{4.2}$$

then the associated invariants are given by

$$I_1 = \sum_i (1+\lambda_i)^2, \quad I_2 = \sum_{i<j} (1+\lambda_i)^2(1+\lambda_j)^2, \quad I_3 = \prod_i (1+\lambda_i)^2, \tag{4.3}$$

non-zeros components of the resulting stress tensor (2nd Piola-Kirchhoff) are constant in this case

$$\Sigma_{ii} = 2\left[\frac{\partial W}{\partial I_1} + \frac{\partial W}{\partial I_2}(I_1 - (1+\lambda_i)^2) + \frac{\partial W}{\partial I_3} I_3 (1+\lambda_i)^{-2}\right], \tag{4.4}$$

the internal forces related to the stress tensor balance with surface forces on the border following the outgoing normal and with moduli equal to $(1+\lambda_i)\Sigma_{ii}$ on each face $x_i = l_i$.
NB : setting homogeneous boundary conditions in displacements along x_i for faces $x_i = 0$ is sufficient for this case.

The RivlinCube test compute the displacements considering the surface forces as an external constant pressure load. We verify at the end of computation that the displacements agree with those given by λ_1, λ_2, λ_3 factors. This description corresponds to the first pass in the `RivlinCube.m` file.

Then 2nd pass compute displacements using the follower pressure instead of the external load (for more details see `fsc`) with $\lambda_1 = \lambda_2 = \lambda_3$. This condition is necessary to ensure continuity in pressure when faces intersect. We also compare displacements computed to those given initially.

4.2 Heat equation

This section was contributed by Bourquin Frédéric and Nassiopoulos Alexandre from *Laboratoire Central des Ponts et Chaussées.*

Problem

Consider a solid occupying a domain Ω in $\mathbb{R}^3$ and let $\partial\Omega$ be its boundary. Inside the solid, the steady state temperature distribution $\theta(x)$ is the solution of the heat equation:

$$-\,div(\mathbf{K}\,grad\,\theta) = f \tag{4.5}$$

where $x = (x_1, x_2, x_3) \in \Omega$ is the space variable, $f = f(x)$ denotes the distributed heat source inside the structure, $\rho = \rho(x)$ the mass density, $c = c(x)$ the heat

capacity and $\mathbf{K}$ the conductivity tensor of the material. The tensor $\mathbf{K}$ is symmetric, positive definite, and is often taken as diagonal. If conduction is isotropic, one can write $\mathbf{K} = k(x)Id$ where $k(x)$ is called the (scalar) conductivity of the material. In this case, (4.5) becomes

$$-\,div(k\,grad\,\theta) = f \tag{4.6}$$

The system is subject to an external temperature $\theta_{ext}(x)$ and a heat flux $g(x)$ along its boundary. The interactions with the surrounding medium can be represented by three kinds of boundary conditions:

- Prescribed temperature (Dirichlet condition, also called boundary condition of first kind)

$$\theta = \theta_{ext} \quad on \quad \partial\Omega \tag{4.7}$$

- Prescribed heat flux (Neumann condition, also called boundary condition of second kind)

$$(\mathbf{K}grad\,\theta)\cdot\vec{n} = g \quad on \quad \partial\Omega \tag{4.8}$$

- Exchange and heat flux (Fourier-Robin condition, also called boundary condition of third kind)

$$(\mathbf{K}grad\,\theta)\cdot\vec{n} + \alpha(\theta - \theta_{ext}) = g \quad on \quad \partial\Omega \tag{4.9}$$

where $\alpha = \alpha(x|_{\partial\Omega}) \geq 0$ denotes the heat exchange coefficient, and $\vec{n}$ the unit outer normal vector to Ω along $\partial\Omega$.

Note that both Dirichlet and Neumann conditions can be viewed as special cases of Fourier-Robin conditions when α tends to infinity and zero respectively. Hence, to compute the solution of (4.5) with a Dirichlet (resp. Neumann) condition, just solve (4.5) with a Fourier-Robin condition where α assumes a very large value (resp. α vanishes).

Variational form

Let us consider the steady-state heat equation with Fourier-Robin boundary conditions for the three-dimensional case.

$$\begin{cases} -div(\mathbf{K}grad\,\theta) = f & in \quad \Omega \\ (\mathbf{K}grad\,\theta)\cdot\vec{n} + \alpha(\theta - \theta_{ext}) = g & on \quad \partial\Omega \end{cases} \tag{4.10}$$

This equation can be put into variational form:

$$\left\{\begin{array}{c} given \quad f \in L^2(\Omega),\, g \in L^2(\partial\Omega) \quad and \quad \theta_{ext} \in H^{\frac{1}{2}}(\partial\Omega), \\ find \quad \theta \in H^1(\Omega) \quad such \quad that \\ \\ \displaystyle\int_\Omega (\mathbf{K} grad\,\theta)(grad\, v)\, dx + \int_{\partial\Omega} \alpha\theta v\, d\gamma = \\ \displaystyle\int_\Omega f v\, dx + \int_{\partial\Omega} g v\, d\gamma + \int_{\partial\Omega} \alpha\theta_{ext} v\, d\gamma \\ \\ \forall v \in H^1(\Omega) \end{array}\right. \tag{4.11}$$

This problem is well-posed whenever α assumes a strictly positive value on a part of the boundary of positive measure (area in 3D), as a consequence of Poincaré-Friedrichs' inequality. The Neumann problem corresponding to the case when the heat exchange coefficient α vanishes identically is more tricky. A solution exists iff the compatibility condition is satisfied. In this case the temperature θ is determined up to an additive constant. If the compatibility condition is not satisfied, solving (4.11) with a small positive value of α is possible, but no convergence occurs when $\alpha \to 0$. In the same way, for any finite value of α, problem (4.11) has a solution even is θ_{ext} is *not* continuous along the boundary. But in this case the solution may not converge when α tends to infinity. Recall that a step function does not belong to the space $H^{\frac{1}{2}}(\mathbb{R})$ and hence is not admissible as a boundary temperature.

The choice of the functional spaces is made to ensure the well-posedness of the variational problem. Note that the regularity of the data is not optimal, but optimal regularity assumptions lie beyond the scope of this documentation.

Test case

Consider a solid square prism of dimensions L_x, L_y, L_z in the three directions (Ox), (Oy) and (Oz) respectively. The solid is made of homogeneous isotropic material, and its conductivity tensor thus reduces to a constant k. The steady state temperature distribution is then given by:

$$-k\Delta\theta = f \quad in \quad \Omega = [0, L_x] \times [0, L_y] \times [0, L_z] \tag{4.12}$$

In what follows, let $\Gamma_i (i = 1..6, \cup_{i=1}^{6} \Gamma_i = \partial\Omega)$ denote each of the 6 faces of the solid. The solid is subject to the following boundary conditions:

- $\Gamma_1\,(x = 0)$: Fourier-Robin condition

$$-\frac{\partial\theta(0, y, z)}{\partial x} + \alpha\theta(0, y, z) = g_1 \tag{4.13}$$

- $\Gamma_2\,(x = L_x)$: Dirichlet condition

$$\theta(L_x, y, z) = \theta_{ext} \tag{4.14}$$

- $\Gamma_3\,(y = 0)$: Fourier-Robin condition

$$-\frac{\partial\theta(x, 0, z)}{\partial y} + \alpha\theta(x, 0, z) = g(x) \tag{4.15}$$

- $\Gamma_4\,(y = L_y)$: Fourier-Robin condition

$$\frac{\partial\theta(x, L_y, z)}{\partial y} + \alpha\theta(x, L_y, z) = g(x) \tag{4.16}$$

- $\Gamma_5\,(z = 0)$: Fourier-Robin condition

$$-\frac{\partial\theta(x, y, 0)}{\partial z} + \alpha\theta(x, y, 0) = g(x) \tag{4.17}$$

- $\Gamma_6\,(z = L_z)$: Fourier-Robin condition

$$\frac{\partial\theta(x, y, L_z)}{\partial z} + \alpha\theta(x, y, L_z) = g(x) \tag{4.18}$$

In above expressions, f is a constant uniform internal heat source, θ_{ext} a constant external temperature at $x = L_x$, $g_1 = \alpha\theta_{ext} + \dfrac{\alpha f L_x^2}{2k}$ a constant and $g(x) = -\dfrac{\alpha f}{2k}x^2 + g_1$. The variational form of equation (4.12) thus reads:

$$k\int_\Omega \nabla\theta\nabla v\,dx + \int_{\cup_{i=3}^6\Gamma_i\,\cup\,\Gamma_1} \alpha\theta v\,d\gamma = \int_{\Gamma_1} g_1 v\,d\gamma + \int_{\cup_{i=3}^6\Gamma_i} gv\,d\gamma + \int_\Omega fv\,dx \tag{4.19}$$

$\forall v \in H_D^1(\Omega)$, together with a Dirichlet condition (fixed DOFs) in Γ_2:

$$\theta|_{\Gamma_2} = \theta_{ext}$$

$H_D^1(\Omega)$ denotes the space of functions in $H^1(\Omega)$ that satisfy the Dirichlet condition. Note that the second term of the left-hand side of equation (4.19) is defined implicitly in the code when assigning the material properties to each surface element group (`femesh` command).

The problem can be solved by the method of separation of variables. It admits the solution:

$$\theta(x, y, z) = -\frac{f}{2k}x^2 + \theta_{ext} + \frac{fL_x^2}{2k} = \frac{g(x)}{\alpha}$$

The OpenFEM model for this example can be found in `demo/heat_equation`.

Numerical application

Assume

$$\begin{array}{ll} k = 400 & f = 40 \\ \alpha = 1 & \theta_{ext} = 20 \\ L_x = 10 & g(x) = 25 - \dfrac{x^2}{20} \\ L_y = 5 & g_1 = 25 \\ L_z = 4 & \end{array} \tag{4.20}$$

Then the solution of the problem is a parabolic distribution along the x-axis:

$$\theta(x, y, z) = 25 - \frac{x^2}{20}$$

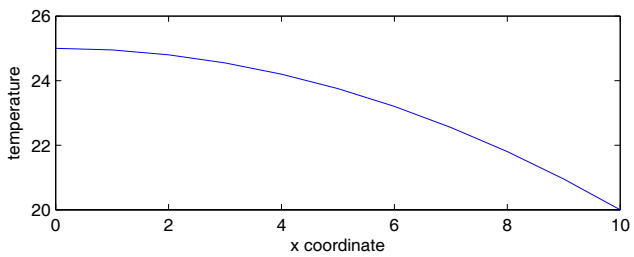


Figure 4.1: Temperature distribution along the x-axis

5

Developer information

This chapter gives a detailed description of the formats used for variables and data structures. This information is grouped here and hypertext reference is given in the HTML version of the manual.

5.1 Nodes

5.1.1 Node matrix

Nodes are characterized using the convention of Universal files. `model.Node` and `FEnode` are node matrices. A node matrix has seven columns. Each row of gives

```
NodeId PID DID GID x y z
```

where `NodeId` are node numbers (positive integers with no constraint on order or continuity), `PID` and `DID` are coordinate system numbers for position and displacement respectively (zero or any positive integer), `GID` is a node group number (zero or any positive integer), and `x y z` are the coordinates . For cylindrical coordinate systems, coordinates represent `r teta z` (radius, angle in degrees, and z axis value). For spherical coordinates systems, they represent `r teta phi` (radius, angle from vertical axis in degrees, azimuth in degrees). For local coordinate system support see section 5.1.2.

A simple line of 10 nodes along the x axis could be simply generated by the command

```
node = [[1:10]' zeros(10,3) linspace(0,1,10)'*[1 0 0]];
```

For other examples take a look at the finite element related demonstrations (see section 3.6) and the mesh handling utility `femesh`.

The **only restriction** applied to the `NodeId` is that they should be positive integers. The earlier limit of `round((2^31-1)/100)` $\approx$ `21e6` is no longer applicable.

In many cases, you will want to access particular nodes by their number. The standard approach is to create a reindexing vector called `NNode`. Thus the commands

```
NNode=[];NNode(node(:,1))=1:size(node,1);
Indices_of_Nodes = NNode(List_of_NodeId)
```

give you a simple mechanism to determine the indices in the `node` matrix of a set of nodes with identifiers `List_of_NodeId`. The `femesh FindNode` commands provide tools for more complex selection of nodes in a large list.

5.1.2 Coordinate system handling

Local coordinate systems are stored in a `model.bas` field described in the `basis` reference section. Columns 2 and 3 of `model.Node` define respectively coordinate system numbers for position and displacement.

feplot, fe_mk, rigid, ... now support local coordinates. feutil does when the model is discribed by a data structure with the .bas field. femesh assumes you are using global coordinate system obtained with

```
[FEnode,bas] = basis(model.Node,model.bas)
```

To write your own scripts using local coordinate systems, it is useful to know the following calls:

[node,bas,NNode]=feutil('getnodebas',model) returns the nodes in global coordinate system, the bases bas with recursive definitions resolved and the reindexing vector NNode.

The command

```
cGL=basis('trans l',model.bas,model.Node,model.DOF)
```

returns the local to global transformation matrix.

5.2 Model description matrices

A *model description matrix* describes the model elements. model.Elt and FEelt are, for example, model description matrices. The declaration of a finite element model is done through the use of element groups stacked as rows of a model description matrix elt and separated by header rows whose first element is Inf in Matlab or %inf in Scilab and the following are the ASCII values for the name of the element. In the following, Matlab notation is used. Don't forget to replace Inf by %inf in Scilab.
For example a model described by

```
elt = [Inf abs('beam1')                0 0
         1  2  11  12  5              0 0 0
         2  3  11  12  5              0 0 0
       Inf abs('mass1')               0 102
         2  1e2 1e2 1e2  5e-5 5e-5 5e-5   0 ];
```

has 2 groups. The first group contains 2 beam1 elements between nodes 1-2 and 2-3 with material property 11, section property 12, and bending plane containing node 5. The second group contains a concentrated mass on node 2.

Note how columns unused for a given type element are filled with zeros. The 102 declared for the mass corresponds to an element group identification number EGID.

You can find more realistic examples of model description matrices in the demonstrations (see section 3.6).

The general format for **header rows** is

```
[Inf abs('ElementName') 0 opt ]
```

The Inf that mark the element row and the 0 that mark the end of the element name are **required** (the 0 may only be omitted if the name ends with the last column of elt).

For multiplatform compatibility, **element names** should only contain lower case letters and numbers. In any case never include blanks, slashes, ... in the element name. Element names reserved for supported elements are listed in the element reference chapter 6 .

Users can define new elements by creating functions (`.m` or `.mex` in Matlab, `.sci` in Scilab) files with the element name. Specifications on how to create element functions are given in section 5.14.

Element group options *opt* can follow the zero that marks the end of the element name. `opt(1)`, if used, should be the element group identification number `EGID`. In the example, the group of `mass1` elements is this associated to the *EGID* 102. The default element group identification number is its order in the group declaration. Negative `EGID` are ignored in FEM analyzes (display only, test information, ...)

Between group headers, each row describes an element of the type corresponding to the previous header (first header row above the considered row).

The general format for **element rows** is

```
[NodeNumbers MatId ProId EltId OtherInfo]
```

where

- `NodeNumbers` are positive integers which must match a unique `NodeId` identifier in the first column of the node matrix.
- `MatId` and `ProId` are material and element property identification numbers. They should be positive integers matching a unique identifier in the first column of the material `pl` and element `il` property declaration matrices.
- `EltId` are positive integers uniquely identifying each element. The `EltIdFix` command (`femesh` and `feutil` function) returns a model that verifies the unicity constraint.
- `OtherInfo` can for example be the node number of a reference node (`beam1` element). These columns can be used to store arbitrary element dependent information. Typical applications would be node dependent plate thickness, offsets, etc.

Note that the position of `MatId`, `ProId` and `EltId` in the element rows are returned by calls of the form `ind=elem0('prop')` (`elem0` is a generic element name, it can be `bar1`, `hexa8`, ...).

Element property rows are used for assembly by `fe_mk`, display by `feplot`, model building by `femesh`, ...

5.3 Material property matrices

This section describes the low level format for material properties. The actual formats are described under `m_` functions `m_elastic`, `m_piezo`, ... For standard scripts see section 3.6.4.

A material is normally defined as a row in the *material property matrix* `pl`. Such rows give a declaration of the general form `[MatId Type Prop]` with

`MatId`	a positive integer identifying a particular material property.
`Type`	a positive real number built using calls of the form `fe_mat('m_elastic','SI',subtype)`, the `subtype` integer is described in `m_` functions.
`Prop`	as many properties (real numbers) as needed (see `fe_mat`, `m_elastic` for details).

Additional information can be stored as an entry of type `'mat'` in the model stack which has data stored in a structure with at least fields

`.name`	Description of material
`.pl`	a single value giving the `MatId` of the corresponding row in the `pl` matrix
`.unit`	a two character string describing the unit system (see the `fe_mat Unit` and `Convert` commands).
`.type`	the name of the material function handling this particular type of material (for example `m_elastic`).

5.4 Element property matrices

This section describes the low level format for element properties. The actual formats are described under `p_` functions `p_shell`, `p_solid`, `p_beam`, `p_spring`. For and standard scripts see section 3.6.4.

An element property is normally defined as a row in the *element property matrix* `il`. Such rows give a declaration of the general form `[ProId Type Prop]` with

`ProId`	a positive integer identifying a particular element property.
`Type`	a positive real number built using calls of the form `fe_mat('p_beam','SI',1)`, the `subtype` integer is described in the `p_` functions.
`Prop`	as many properties (real numbers) as needed (see `fe_mat`, `p_solid` for details).

Additional information can be stored as an entry of type `'pro'` in the model stack which has data stored in a structure with fields

`.name`	description of property.
`.il`	a single value giving the `ProId` of the corresponding row in the `il` matrix
`.unit`	a two character string describing the unit system (see the `fe_mat Unit` and `Convert` commands).
`.type`	the name of the property function handling this particular type of element properties (for example `p_beam`).

The handling of a particular type of constants should be fully contained in the `p_*` function. The meaning of various constants should be defined in the help and TEX documentation. The subtype mechanism can be used to define sereral behaviors

of the same class. The generation of the `integ` and `constit` vectors should be performed through a `BuildConstit` call that is the same for a full family of element shapes. The generation of `EltConst` should similarly be identical for an element family.

5.5 DOF definition vector

OpenFEM keeps track of the meaning of each Degree of Freedom (DOF) trough DOF definition vectors (see details below). As `mdof` keeps track of the meaning of different DOFs, `fe_c` can be used manipulate incomplete and unordered DOF sequences. This is used for boundary condition manipulations, renumbering, ...

OpenFEM distinguishes nodal and element DOFs.

Nodal DOFs are described as a single number of the form `NodeId.DofId` where `DofId` is an integer between `01` and `99`. For example DOF 1 of node 23 is described by `23.01`. By convention

- DOFs `01` to `06` are, in the following order u, v, w (displacements along the global coordinate axes) and θ_u, θ_v, θ_w (rotations along the same directions)
- DOFs `07` to `12` are, in the following order $-u$, $-v$, $-w$ (displacements along the reversed global coordinate axes) and $-\theta_u$, $-\theta_v$, $-\theta_w$ (rotations along the same directions). This convention is used in test applications where measurements are often made in those directions and not corrected for the sign change. It should not be used for finite element related functions which may not all support this convention.

While these are the only mandatory conventions, other typical DOFs are `.19` pressure, `.20` temperature, `.21` voltage, `.22` magnetic field.

In a small shell model, all six DOFs (translations and rotations) of each node would be retained and could be stacked sequentially node by node. The DOF definition vector `mdof` and corresponding displacement or load vectors would thus take the form

$$\texttt{mdof} = \begin{bmatrix} \texttt{1.01} \\ \texttt{1.02} \\ \texttt{1.03} \\ \texttt{1.04} \\ \texttt{1.05} \\ \texttt{1.06} \\ \vdots \end{bmatrix}, \texttt{q} = \begin{bmatrix} u_1 & u_2 & \\ v_1 & v_2 & \\ w_1 & w_2 & \\ \theta_{u1} & \theta_{u2} & \cdots \\ \theta_{v1} & \theta_{v2} & \\ \theta_{w1} & \theta_{w2} & \\ \vdots & & \ddots \end{bmatrix} \text{ and } \texttt{F} = \begin{bmatrix} F_{u1} & F_{u2} & \\ F_{v1} & F_{v2} & \\ F_{w1} & F_{w2} & \\ M_{u1} & M_{u2} & \cdots \\ M_{v1} & M_{v2} & \\ M_{w1} & M_{w2} & \\ \vdots & & \ddots \end{bmatrix}$$

Typical vectors and matrices associated to a DOF definition vector are

- **modes** resulting from the use of `fe_eig` .
- **input and output shape matrices** which describe how forces are applied and sensors are placed (see `fe_c`, `fe_load`).

- **system matrices** : mass, stiffness, etc. assembled by `fe_mk`.
- **FRF** test data. If the position of sensors is known, it can be used to animate experimental deformations (see `feplot`).

Note that, in Matab version, the functions `fe_eig` and `fe_mk`, for models with more than 1000 DOFs, renumber DOF internally so that you may not need to optimize DOF numbering yourself. In such cases though, `mdof` will not be ordered sequentially as shown above.

Element DOFs are described as a single number of the form `-EltId.DofId` where `DofId` is an integer between `001` and `999`. For example DOF 1 of the element with ID `23001` is described by `-23001.001`. Element DOFs are typically only used by superelements . Due to the use of integer routines for indexing operations, you cannot define element DOFs for elements with and `EltId` larger than 2 147 484.

5.6 FEM model structure

Finite element simulations are best handled using standard data structures supported by *OpenFEM*. The two main data structures are `model` which contains information needed to specify a FEM problem, and `DEF` which stores a solution.

Finite element models are described by their topology (nodes, elements and possibly coordinate systems), their properties (material and element). Computations performed with a model are further characterized by a case as illustrated in section 3.7 and detailed in section 5.7.

Data structures describing finite element models have the following standardized fields, where only nodes and elements are always needed.

`.bas`	local coordinate system definitions
`.cta`	sensor observation matrix.
`.copt`	solver options. This field is likely to disappear in favor of defaults in `sdtdef`.
`.DOF`	`DOF definition vector` for the matrices of the model. Boundary conditions can be imposed using cases.
`.Elt`	elements. This field is **mandatory**.
`.file`	Storage file name.
`.il`	element property description matrix. Can also be stored as `'pro'` entries in the `Stack`.
`.K{`i`}`	cell array of constant matrices for description of model as a linear combination. Indices i match definitions in `.Opt(2,:)` and `.Opt(3,:)`. Should be associated with a `.Klab` field giving a string definition of each matrix.
`.mind`	element matrix indices.
`.Node`	nodes. This field is **mandatory**.
`.Opt`	options characterizing models that are to be used as superelements
`.pl`	material property description matrix. Can also be stored as `'mat'` entries in the `Stack`.
`.Patch`	Patch face matrix.
`.Stack`	A cell array containing optional properties further characterizing a finite element model. See `stack_get` for how to handle the stack and the next section for a list of standardized entries.
`.TR`	projection matrix.
`.unit`	main model unit system (see `fe_mat Convert` for a list of supported unit systems and the associated two letter codes). Specifying this field let you perform conversion from materials defined in `US` system unit from the GUI.
`.wd`	working directory

Obsolete fields are `.Ref` Generic coordinate transformation specification, `.tdof` test DOF field (now in `SensDof` entries).

5.7 FEM stack and case entries

Various information are stored in the `model.Stack` field.

Currently supported entries in the stack are

case	defines a case : boundary conditions, loading, ...
curve	curve to be used for simulations (see fe_curve)
info	non standard information used by solvers or meshing procedures (see below)
mat	defines a material entry
SE	defines a superelement entry.
sel	defines an element selection
seln	defines a node selection. Typically a structure with fields .ID giving the reference number and .data giving either node numbers or a node selection command.
set	defines a set. Typical sets are lists of node or element identifiers (structure with fields .ID and .data) or edge/face references (structures with fields .ID giving the reference number and .data with two columns giving EltId and edge number, as detailed in integrules. Sets can be used to define loaded surfaces. The optional field .type can specify the nature of the set: NodeId, EltId, FaceId or DOF. feutil AddSet commands let you define a set from a selection.
pro	defines an element property entry

Currently reserved names for info entries are

DefaultZeta	value to be used as default modal damping ratio (viscous damping). The default loss factor if needed is taken to be twice that value.
EigOpt	gives real eigenvalue solver options (see fe_eig).
FluidEta	Default loss factor for use in vibroacoustic fluid computations
Freq	Frequencies given as a structure with field .X with frequency values and .ID a integer identifier.
NewNodeFrom	integer giving the next NodeId to be used when adding nodes to the model (used by some commands of feutil).
Omega	rotation vector used for rotating machinery computations (see fe_cyclic).
OrigNumbering	original node numbering (associated with feutil Renumber command). Two int32 columns giving original and new node numbers.
TimeOpt	gives time solver options (see fe_time).

A case defines finite element boundary conditions, applied loads, physical parameters, ... The associated information is stored in a case data structure with fields

Case.Stack	list of boundary conditions, constraints, parametric design point, and loading cases that need to be considered. A table of accepted entries is given under fe_case. Each row gives {Type,Name,data}.
Case.T	basis of subspace verifying fixed boundary conditions and constraints.
Case.DOF	DOF definition vector describing the columns of T, the rows of T are described by the .DOF field of the model.

The various cases are then stored in the .Stack field of the model data structure (this is done by a call to fe_case).

5.8 FEM result data structure

Deformations resulting from finite element computations (`fe_eig`, `fe_load`, ...) are described by a structure with fields

`.def`	deformations ($NDOF$ by $NDef$ matrix)
`.DOF`	`DOF definition vector`
`.data`	(optional) matrix of numbers characterizing the content of each deformation (frequency, time step, ...)
`.opt`	options
`.fun`	function description `[Model Analysis Field Signification Format]` (see `xfopt _funtype`)
`.lab`	(optional) cell array of strings characterizing the content of each deformation.
`.label`	string describing the content
`.scale`	(optional) string describing the content

5.9 Curves and data sets

Curves are used to specify inputs (for time or frequency domain simulation) and store results from simulations. They can be stored as entries `{'curve', Name, data}` in the model stack or in the case of inputs in the `load.curve` cell array.

A curve can be used to define a time (or frequency) dependent load $\{F\} = [B]\{u\}$. $[B]$ defines the spatial distribution of the load on DOFs and its unit is the same as F. $[B]$ is defined by a `DOFLoad` entry in the Case. $\{u\}$ defines the time (or frequency) dependency as a unitless curve. There should be as many curves as columns in the matrix of a given load `def`. If a single curve is defined for a multi-load entry, it will affect all the loads of this entry.

As an illustration, let us consider ways to define a time dependent load by defining a `.curve` field in the load data structure. This field may contain a string referring to an existing curve (name is `'input'` here)

```
model=fe_time('demo bar');fe_case(model,'info')

% redefine curve
model=fe_curve(model,'set','input','TestStep 1e-3');
% have load reference that curve
model=fe_case(model,'setcurve','Point load 1','input');

TimeOpt=fe_time('timeopt newmark .25 .5 0 1e-4 100');
model=stack_set(model,'info','TimeOpt',TimeOpt);
def=fe_time(model); feplot(model,def)
```

or a data structure generated by `fe_curve`

```
model=fe_time('demo bar');fe_case(model,'info')
```

```
model=fe_case(model,'remove','fd'); % loads at both ends
data=struct('DOF',[1.01;2.01],'def',1e6*eye(2),...
            'curve',{{'test ricker 10e-4 1',...
                      'test ricker 20e-4 1'}});
model = fe_case(model,'DOFLoad','Point load 1',data);

TimeOpt=fe_time('timeopt newmark .25 .5 0 1e-4 100');
model=stack_set(model,'info','TimeOpt',TimeOpt);
def=fe_time(model); feplot(model,def)
```

A curve is a `data` structure with fields

`.ID`	identification and type of the curve
`.X`	axis data. A cell array with as many entries as dimensions of `.Y`. Contents of each cell can be a vector (for example vector of frequencies or time steps), a cell array describing data vectors in `.Y` (for example response labels) (obsolete) x-axis data.
`.Xlab`	A cell array of strings giving the meaning of each entry in `.X`
`.Y`	response data. If a matrix rows correspond to `.X` values and columns are called *channels*
`.Z`	(obsolete) should be stored as a third dimension in `.Y` and filling of `.X{3}`.
`.data`	a matrix with one row per channel (column of `.Y`). This is used to store DOF information for responses, pole information for modes, ...
`.*unit`	(obsolete) a cell array with three columns giving `label` the meaning of the x axis, `ulabel` the unit label for the x axis, and `typ` four values giving the type number, followed by length, force, temperature and time unit exponents (these are used for automated unit system conversion). Typical fields can be generated with `fe_curve('DatatypeCell','Time')`. This information should now be stored as entries in `.X{3}`
`.name`	name of the curve
`.type`	`'fe_curve'`
`.unit`	unit system of the curve (see `fe_mat convert`)
`.Interp`	optional interpolation method. Available interpolations are `linear`, `log` and `stair`
`.Extrap`	optional extrapolation method. Available extrapolations are `flat`, `zero` and `exp`
`.PlotInfo`	type of plotting

5.10 DOF selection

`fe_c` is the general purpose function for manipulating DOF definition vectors. It is called by many other functions to select subsets of DOFs in large DOF definition

vectors. DOF selection is very much related to building an observation matrix `c`, hence the name `fe_c`.

For DOF selection, `fe_c` arguments are the reference DOF vector `mdof` and the DOF selection vector `adof`. `adof` can be a standard DOF definition vector but can also contain wild cards as follows

`NodeId.0`	means all the DOFs associated to node `NodeId`
`0.DofId`	means `DofId` for all nodes having such a DOF
`-EltN.0`	means all the DOFs associated to element `EltId`

Typical examples of DOF selection are

`ind = fe_c(mdof,111.01,'ind');` returns the position in `mdof` of the x translation at node 111. You can thus extract the motion of this DOF from a vector using `mode(ind,:)`. Note that the same result would be obtained using an output shape matrix in the command `fe_c(mdof,111.01)*mode`.

```
model = fe_mk(model,'FixDOF','2-D motion',[.03 .04 .05])
```

assembles the model but only keeps translations in the xy plane and rotations around the z axis (DOFs `[.01 .02 .06]'`). This is used to build a 2-D model starting from 3-D elements.

The `feutil FindNode` commands provides elaborate node selection tools. Thus `femesh('findnode x>0')` returns a vector with the node numbers of all nodes in the standard global variable `FEnode` that are such that their x coordinate is positive. These can then be used to select DOFs, as shown in the section on boundary conditions section 5.13. Node selection tools are described in the next section.

5.11 Node selection

`feutil FindNode` supports a number of node selection criteria that are used by many functions. A node selection command is specified by giving a string command (for example `'GroupAll'`, or the equivalent cell array representation described at the end of this section) to be applied on a model (nodes, elements, possibly alternate element set).

Output arguments are the numbers `NodeId` of the selected nodes and the selected nodes `node` as a second optional output argument. The basic commands are

- `[NodeId,node]=feutil(['findnode ...'],model)` or `node=feutil(['getnode ...'],model)`
  this command applies the specified node selection command to a `model` structure. For example, `[NodeId,node] = feutil('findnode x==0',model);` selects the nodes in `model.Node` which first coordinate is null.
- `[NodeId,node]=femesh(['findnode ...'])`
  this command applies the specified node selection command to the standard global matrices `FEnode`, `FEelt`, `FEel0`, ... For example, `[NodeId,node] = femesh('findnode x==0');` selects the node in `FEnode` which first coordinate is null.

Accepted selectors are

`GID` *i*	selects the nodes in the node group *i* (specified in column 4 of the node matrix). Logical operators are accepted.
`Group` *i*	selects the nodes linked to elements of group(s) *i* in the main model. Same as `InElt{Group` *i*`}`
`Groupa` *i*	selects nodes linked to elements of group(s) *i* of the alternate model
`InElt{`*sel*`}`	selects nodes linked to elements of the main model that are selected by the element selection command `sel`.
`NodeId >`*i*	selects nodes selects nodes based relation of `NodeId` to integer *i*. The logical operator `>`, `<`, `>=`, `<=`, `~=`, or `==` can be omitted (the default is then `==`). `feutil('findnode 1 2',model)` interprets the values as `NodeId` unless three values are given (then intrepreted as `x y z`). `feutil('findnode',model,IdList)` should then be used.
`NotIn{`*sel*`}`	selects nodes not linked to elements of the main model that are selected by the element selection command `sel`.
`Plane ==` *i nx ny nz*	selects nodes on the plane containing the node number *i* and orthogonal to the vector `[`*nx ny nz*`]`. Logical operators apply to the oriented half plane. *i* can be replaced by string `o xo yo zo` specifying the origin.

`rad <=`*r x y z*	selects nodes based on position relative to the sphere specified by radius *r* and position *x y z* node or number *x* (if *y* and *z* are not given). The logical operator `>`, `<`, `>=`, `<=` or `==` can be omitted (the default is then `<=`).
`Setname` *name*	finds nodes based on a set defined in the model stack. Note that the name must not contain blanks or be given between double quotes `"name"`. Set can be a `NodeId` or even an `EltId` or `FaceId` set.
`x>`*a*	selects nodes such that their x coordinate is larger than *a*. `x y z r` (where the radius `r` is taken in the xy plane) and the logical operators `>`, `<`, `>=`, `<=`, `==` can be used. Expressions involving other dimensions can be used for the right hand side. For example `r>.01*z+10`.
x y z	selects nodes with the given position. If a component is set to `NaN` it is ignored. Thus `[0 NaN NaN]` is the same as `x==0`.

Element selectors `EGID`, `EltId`, `EltName`, `MatId` and `ProId` are interpreted as `InElt` selections.

Command modifier `epsl` *value* can be used to give an evaluation tolerance for equality logical operators.

Different selectors can be chained using the logical operations `&` (finds nodes that verify both conditions), `|` (finds nodes that verify one or both conditions). Condition combinations are always evaluated from left to right (parentheses are not accepted).

While the string format is typically more convenient for the user, the reference format for a node selection is really a 4 column cell array :

{	Selector	Operator	Data
Logical	Selector	Operator	Data
}			

The first column gives the chaining between different rows, with `Logical` being either `&`, `|` or a bracket `(` and `)`. The `Selector` is one of the accepted commands for node selection (or element selection if within a bracket). The `operator` is a logical operator `>`, `<`, `>=`, `<=`, `~=`, or `==`. The `data` contains numerical or string values that are used to evaluate the operator. Note that the meaning of `~=` and `==` operators is slightly different from base MATLAB operators as they are meant to operate on sets.

The `feutil findnodestack` command returns the associated cell array rather than the resulting selection.

5.12 Element selection

`feutil` supports a number of element selection criteria that are used by many functions. An element selection command is specified by giving a string command (for example `'GroupAll'`) to be applied on a model (nodes, elements, possibly alternate element set).

Basic commands are :

- `[eltind,elt] = feutil('findelt selector',model);`
  or `elt = feutil('selelt selector',model);` this command applies the specified element selection command to a `model` structure. For example, `[eltind,selelt] = feutil('findelt eltname bar1',model)` selects the elements in `model.Elt` which type is `bar1`.
- `[eltind,elt] = feutil('findelt selector',model);`
  this command applies the specified element selection command to the standard global matrices `FEnode`, `FEelt`, `FEel0`, ... For example, `[eltind,selelt] = femesh('findelt eltname bar1')` selects the elements in `FEelt` which type is `bar1`.

Output arguments are `eltind` the selected elements indices in the element description matrix and `selelt` the selected elements.

Accepted selectors are

`EltId` *i*	finds elements with identificators *i* in `FEelt`. Operators accepted.
`EltInd` *i*	finds elements with indices *i* in `FEelt`. Operators accepted.
`EltName` *s*	finds elements with element name *s*. `EltName flui` will select all elements with name starting with `flui`. `EltName = flui` will select all elements with name not starting with `flui`. One can select superelements from their name using `EltName SE:`*SEName*.
`SetName` *s*	finds elements in element set named *s*. See example in `feutil AddSet`.
`EGID` *i*	finds elements with element group identifier *i*. Operators accepted.
`Facing >` *cos* *x y z*	finds topologically 2-D elements whose normal projected on the direction from the element CG to *x y z* has a value superior to *cos*. Inequality operations are accepted.
`Group` *i*	finds elements in group(s) *i*. Operators accepted.
`InNode` *i*	finds elements with all nodes in the set *i*. Nodes numbers in *i* can be replaced by a string between braces defining a node selection command. For example `femesh('find elt withnode {y>-230 & NodeId>1000}')`.
`MatId` *i*	finds elements with `MatId` equal to *i*. Relational operators are also accepted (`MatId =1:3`, ...).
`ProId` *i*	finds elements with `ProId` equal to *i*. Operators accepted.
`SelEdge` *type*	selects the external edges (lines) of the currently selected elements (any element selected before the `SelEdge` selector), any further selector is applied on the model resulting from the `SelEdge` command rather than on the original model. Type `g` retains inter-group edges. `m` retains inter-material edges. Type `p` retains inter-property edges. `all` retains all edges. The `MatId` for the resulting model identifies the original properties of each side of the edge.
`SelFace` *type*	selects the external faces (surfaces) of the currently selected elements (see more details under `SelEdge`).
`WithNode` *i*	finds elements with at least one node in the set *i*. `i` can be a list of node numbers. Replacements for *i* are accepted as above.
`WithoutNode`*i*	finds elements without any of the nodes in the set *i*. `i` can be a list of node numbers. Replacements for *i* are accepted as above.

Different selectors can be chained using the logical operations `&` (finds elements that verify both conditions), `|` (finds elements that verify one or both conditions). `femesh('FindEltGroup 1:3 & with node 1 8')` for example. Condition combinations are always evaluated from left to right (parentheses are not accepted).

Command modifier `epsl` *value* can be used to give an evaluation tolerance for equality logical operators.

Numeric values to the command can be given as additional `femesh` arguments. Thus the command above could also have been written `femesh('findelt group & withnode',1:3,[1 8])`.

5.13 Constraint and fixed boundary condition handling

5.13.1 Theory and basic example

`rigid` links, `FixDof`, `MPC` entries, symmetry conditions, continuity constraints in CMS applications, ... all lead to problems of the form

$$\begin{array}{c}\left[Ms^2+Cs+K\right]\{q(s)\}=[b]\{u(s)\}\\ \{y(s)\}=[c]\{q(s)\}\\ [c_{int}]\{q(s)\}=0\end{array} \tag{5.1}$$

The linear constraints $[c_{int}]\{q(s)\}=0$ can be integrated into the problem using Lagrange multipliers or constraint elimination. Elimination is done by building a basis T for the kernel of the constraint equations, that is such that

$$\text{range}([T]_{N\times(N-NC)})=\ker([c_{int}]_{NS\times N}) \tag{5.2}$$

Solving problem

$$\begin{array}{c}\left[T^TMTs^2+T^TCTs+T^TKT\right]\{q_R(s)\}=\left[T^Tb\right]\{u(s)\}\\ \{y(s)\}=[cT]\{q_R(s)\}\end{array}$$

is then strictly equivalent to solving (5.1).

The basis T is generated using `[Case,model.DOF]=fe_case(model,'gett')` where `Case.T` gives the T basis and `Case.DOF` describes the active or master DOFs (associated with the columns of T) while `model.DOF` describes the full list of DOFs.

The assembly of unconstrained M, ... or constrained T^TMT matrices can be controlled with appropriate options in `fe_mknl`, `fe_load`, ... Typically a `NoT` string is added to the command.

For the two bay truss example, can be written as follows :

```
femesh('reset');
model2 = femesh('test 2bay');
model2=fe_case(model, ...          % defines a new case
  'FixDof','2-D motion',[.03 .04 .05]', ...  % 2-D motion
  'FixDof','Clamp edge',[1 2]');             % clamp edge
Case=fe_case('gett',model2)  % Notice the size of T and
fe_c(Case.DOF)                % display the list of active DOFs
model2 = fe_mknl(model2)

% Now reassemble unconstrained matrices and verify the equality
% of projected matrices
[m,k,mdof]=fe_mknl(model2,'NoT');

norm(full(Case.T'*m*Case.T-model2.K{1}))
norm(full(Case.T'*k*Case.T-model2.K{2}))
```

5.13.2 Local coordinates

In the presence of local coordinate systems (non zero value of DID in node column 3), the Case.cGL matrix built during the gett command, gives a local to global coordinate transformation

$$\{q_{global}\} = [cGL]\,\{q_{local}\}$$

Master DOFs (DOFs in Case.DOF) are defined in the local coordinate system. As a result, M is expected to be defined in the global response system while the projected matrix $T^T MT$ is defined in local coordinates. mpc constraints are defined using the local basis.

5.13.3 Enforced displacement

For a DofSet entry, one defines the enforced motion in Case.TIn and associated DOFs in Case.DofIn. The DOFs specified in Case.DofIn are then fixed in Case.T.

5.13.4 Low level examples

A number of low level commands (feutil GetDof, FindNode, ...) and functions fe_c can be used to operate similar manipulations to what fe_case GetT does, but things become rapidly complex. For example

```
femesh('reset'); model = femesh('test 2bay');
[m,k,mdof]=fe_mknl(model)

i1 = femesh('findnode x==0');
adof1 = fe_c(mdof,i1,'dof',1);              % clamp edge
adof2 = fe_c(mdof,[.03 .04 .05]','dof',1); % 2-D motion
adof = fe_c(mdof,[adof1;adof2],'dof',2);

ind = fe_c(model.DOF,adof,'ind');
mdof=mdof(ind); tmt=m(ind,ind); tkt=k(ind,ind);
```

Handling multiple point constraints (rigid links, ...) really requires to build a basis T for the constraint kernel. For rigid links the obsolete rigid function supports some constraint handling. The following illustrates restitution of a constrained solution on all DOFs

```
% Example of a plate with a rigid edge
model=femesh('testquad4 divide 10 10');femesh(model)

% select the rigid edge and set its properties
femesh(';selelt group1 & seledge & innode {x==0};addsel');
```

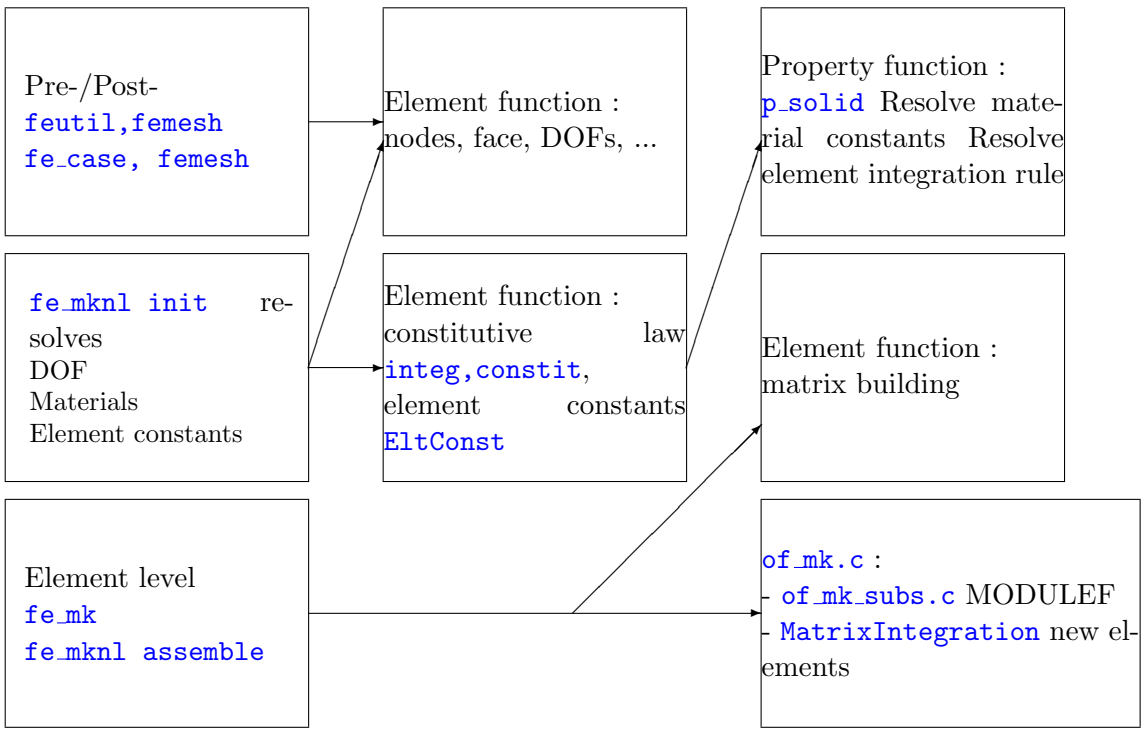
```
femesh('setgroup2 name rigid');
FEelt(femesh('findelt group2'),3)=123456;
FEelt(femesh('findelt group2'),4)=0;
model=femesh;

% Assemble
model.DOF=feutil('getdof',model);% full list of DOFs
[tmt,tkt,mdof] = fe_mknl(model); % assemble constrained matrices
Case=fe_case(model,'gett');      % Obtain the transformation matrix

[md1,f1]=fe_eig(tmt,tkt,[5 10 1e3]); % compute modes on master DOF

def=struct('def',Case.T*md1,'DOF',model.DOF) % display on all DOFs
feplot(model,def); fecom(';view3;ch7')
```

5.14 Creating new elements (advanced tutorial)

In this section one describes the developments needed to integrate a new element function into *OpenFEM*. First, general information about OpenFEM work is given. Then the writing of a new element function is described. And at last, conventions which must be respected are given.

5.14.1 Conventions

Element functions and C functionality



In *OpenFEM*, elements are defined by element functions. Element functions provide different pieces of information like geometry, degrees of freedom, model matrices, ...

OpenFEM functions like the preprocessor `femesh`, the model assembler `fe_mk` or the post-processor `feplot` call element functions for data about elements.

For example, in the assembly step, `fe_mk` analyzes all the groups of elements. For each group, `fe_mk` gets its element type (*bar1*, *hexa8*, ...) and then calls the associated element function.
First of all, `fe_mk` calls the element function to know what is the rigth call form to compute the elementary matrices (`eCall=elem0('matcall')` or `eCall=elem0('call')`, see section 5.14.7 for details). `eCall` is a string. Generally, `eCall` is a call to the element function. Then for each element, `fe_mk` executes `eCall` in order to compute the elementary matrices.

This automated work asks for a likeness of the element functions, in particular for the calls and the outputs of these functions. Next section gives information about element function writing.

Standard names in assembly routines

`cEGI`	vector of element property row indices of the current element group (without the group header)
`constit`	real (`double`) valued constitutive information. The `constit` for each group is stored in `Case.GroupInfo{jGroup,4};`.
`def.def`	vector of deformation at DOFs. This is used for non-linear, stress or energy computation calls that need displacement information.
`EGID`	Element Group Identifier of the current element group (different from `jGroup` if an EGID is declared).
`elt`	model description matrix. The element property row of the current element is given by `elt(cEGI(jElt),:)` which should appear in the calling format `eCall` of your element function.
`ElemF`	name of element function or name of superelement
`ElemP`	parent name (used by `femesh` in particular to allow property inheritance)
`gstate`	real (`double`) valued element state information.
`integ`	`int32` valued constitutive information.
`jElt`	number of the current element in `cEGI`
`jGroup`	number of the current element group (order in the element matrix). `[EGroup,nGroup]=getegroup(elt);` finds the number of groups and group start indices.
`nodeE`	nodes of the current element.
`NNode`	node identification reindexing vector. `NNode(ID)` gives the row index (in the `node` matrix) of the nodes with identification numbers `ID`. You may use this to extract nodes in the `node` matrix using something like `node(NNode(elt(cEGI(jElt),[1 2])),:)` which will extract the two nodes with numbers given in columns 1 and 2 of the current element row (an error occurs if one of those nodes is not in `node`). This can be built using `NNode=sparse(node(:,1),1,1:size(node,1)`.
`pointers`	one column per element in the current group gives.

Case.GroupInfo cell array

The meaning of the columns is as follows

`DofPos Pointers Integ Constit gstate ElMap InfoAtNode EltConst`

`DofPos` `int32` matrix whose columns give the DOF positions in the full matrix of the associated elements. Numbering is C style (starting at 0) and -1 is used to indicate a fixed DOF.

`pointers` `int32` matrix whose columns describe information each element of the group. Pointers has one column per element giving
`[OutSize1 OutSize2 u3 NdNRule MatDes IntegOffset ConstitOffset StateOffset]`
`Outsize1` size of element matrix (for elements issued from MODULEF), zero otherwise.
`MatDes` type of desired output. See the `fe_mk MatType` section for a current list.
`IntegOffset` gives the starting index (first element is 0) of integer options for the current element in `integ`.
`ConstitOffset` gives the starting index (first element is 0) of real options for the current element in `constit`.

`integ` `int32` matrix storing integer values used to describe the element formulation of the group. Meaning depends on element family and should be documented in the element property function (`p_solid BuildConstit` for example).
The nominal content of an `integ` column (as return by the element `integinfo` call) is
`MatId,ProId,NDofPerElt,NNodePerElt,IntegRuleType`
where `integrules(ElemP,IntegRuleType)` is supposed to return the approriate integration rule.

`constit` `double` matrix storing integer values used to describe the element formulation of the group. Meaning depends on element family and should be documented in the element property function (`p_solid BuildConstit` for example).

`gstate` `double` matrix whose columns describe the internal state of each element of the group. By default, column content is stress at integration points ($Nstrain \times Nw$ values). Users are of course free to add any appropriate value for their own elements, a typical application is the storage of internal variables. For an example of `gstate` initialization see `fe_stres` thermal.

`ElMap` `int32` element map matrix used to distinguish between internal and external element DOF numbering (for example : `hexa8` uses all x DOF, then all y ... as internal numbering while the external numbering is done using all DOFs at node 1, then node 2, ...). The element matrix in extrenal sort is given by `k_ext=ke(ElMap)`. `EltConst.VectMap` gives similar reordering information for vectors (loads, ...).

`InfoAtNode` a structure with `.NodePos (int32)` with as many columns as elements in the group giving column positions in a `.data` field. The each row in `.data` corresponds to a field that should be described by a cell array of string in `.lab` for validation purposes.
obsolete : `double` matrix whose **rows** describe information at element nodes (as many columns as nodes in the model).

`EltConst` `struct` used to store element formulation information (integration rule, constitutive matrix topology, etc.) Details on this data structure are given in section 5.14.1.

Element constant data structure

The `EltConst` data structure is used in most newer generation elements implemented in `of_mk.c`. It contains geometric and integration rule properties. The shape information is generated by calls to `integrules`. The formulation information is generated `p_function const` calls (see `p_solid`, `p_heat`, ...).

`.N` $nw \times Nnode$ shape functions at integration points

`.Nr` $nw \times Nnode$ derivative of shape function with respect to the first reference coordinate r

`.Ns` $nw \times Nnode$ derivative of shape function with respect to the second reference coordinate s

`.Nt` $nw \times Nnode$ derivative of shape function with respect to the second reference coordinate t

`.NDN` $Nshape \times nw(1 + Ndim)$ memory allocation to store the shape functions and their derivatives with respect to physical coordinates $[N\ N, x\ N, y\ N, z]$. `of_mk` currently supports the following geometry rules `3` 3D volume, `2` 2D volume, `23` 3D surface, `13` 3D line (see `integrules BuildNDN` for calling formats). Cylindrical and spherical coordinates are not currently supported. In the case of rule `31` (hyperelastic elements), the storage scheme is modified to be $(1 + Ndim) \times Nshape \times nw$ which preserves data locality better.

`.jdet` Nw memory allocation to store the determinant of the jacobian matrix at integration points.

`.bas` $9 \times Nw$ memory allocation to store local material basis. This is in particular used for 3D surface rules where components `6:9` of each column give the normal.

`.Nw` number of integration points (equal to `size(EltConst.N,1)`)

`.Nnode` number of nodes (equal to `size(EltConst.N,2)=size(EltConst.NDN,1)`)

`.xi` $Nnode \times 3$ reference vertex coordinates

`.VectMap` index vector giving DOF positions in external sort. This is needed for RHS computations.

5.14.2 Generic compiled linear and non-linear elements

To improve the ease of development of new elements, OpenFEM now supports a new

category of generic element functions. Matrix assembly, stress and load assembly calls for these elements are fully standardized to allow optimization and generation of new element without recompilation. All the element specific information stored in the `EltConst` data structure.

Second generation volume elements are based on this principle and can be used as examples. These elements also serve as the current basis for non-linear operations.

The adopted logic is to develop families of elements with different toplogies. To implement a family, one needs

- shape functions and integration rules. These are independent of the problem posed and grouped systematically in `integrules`.
- topology, formatting, display, test, ... information for each element. This is the content of the element function (see hexa8, tetra4, ...) .
- a procedure to build the `constit` vectors from material data. This is nominally common to all elements of a given family and is used in `integinfo` element call. For example `p_solid('BuildConstit')`.
- a procedure to determine constants based on current element information. This is nominally common to all elements of a given family and is used in `groupinit` phase (see `fe_mk`). The `GroupInit` call is expected to generate an `EltConst` data structure, that will be stored in the last column of `Case.GroupInfo`. For example `hexa8 constants` which calls `p_solid('ConstSolid')`.
- a procedure to build the element matrices, right hand sides, etc. based on existing information. This is compiled in `of_mk MatrixIntegration` and `StressObserve` commands. For testing/development purposes is expected that for `sdtdef('diag',12)` an `.m` file implementation in `elem0.m` is called instead of the compiled version.

The following sections detail the principle for linear and non-linear elements.

5.14.3 What is done in the element function

Most of the work in defining a generic element is done in the element property function (for initializations) and the compile `of_mk` function. You do still need to define the commands

- `integinfo` to specify what material property function will be called to build `integ`, `constit` and `elmap`. For example, in `hexa8`, the code for this command command is

```
if comstr(Cam,'integinfo')
 %constit integ,elmap                          ID,pl,il
 [out,out1,out2]= ...
  p_solid('buildconstit',[varargin{1};24;8],varargin{2},varargin{3});
```

input arguments passed from fe_mknl are ID a unique pair of MatId and ProId in the current element group. pl and il the material and element property fields in the model. Expected outputs are constit, integ and elmap, see Case.GroupInfo. Volume elements hexa8, q4p, ... are topology holders. They call p_solid BuildConstit which in turn calls as another property function as coded in the type (column two of il coded with fe_mat('p_fun','SI',1)). When another property funciton is called, it is expected that constit(1:2)=[-1 TypeM] to allow propagation of type information to parts of the code that will not analyze pl.

- constants to specify what element property function will be called to initialize EltConst data structure and possibly set the geometry type information in pointers(4,:). For example, in hexa8, the code for this command command is

```
...
elseif comstr(Cam,'constants')
  integ=varargin{2};constit=varargin{3};
  if nargin>3; [out,idim]=p_solid('const','hexa8',integ,constit);
  else; p_solid('constsolid','hexa8',[1 1 24 8],[]);return;
  end
  out1=varargin{1};out1(4,:)=idim; % Tell of_mk('MatrixInt') this is IDIM
...
```

  input arguments passed from fe_mknl are pointers,integ,constit the output arguments are EltConst and a modified pointers where row 4 is modified to specify a 3D underlying geometry.

  If constit(1:2)=[-1 TypeM] p_solid calls the appropriate property function.

  For elements that have an internal orientation (shells, beams, etc.) it is expected that orientation maps are built during this command (see beam1t, ...).

- standard topology information (commands node, dof, prop, line, patch, face, edge, parent) see section 5.14.7.

hexa8 provides a clean example of what needs to be done here.

5.14.4 What is done in the property function

p_fcn Commands specific to p_* are associated to the implementation of a particular physical formulation for all topologies.

BuidConstit

As shown in section 5.14.1 and detailed under fe_mknl the FEM initialization phase needs to resolve

- constitutive law information from model constants (elem0 integinfo call to the element functions, which for all topology holder elements is forwarded to p_solid BuildConstit)
- and to fill-in integration constants and other initial state information (using groupinit to generate the call and constant build the data).

Many aspects of a finite element formulation are independent of the supporting topology. Element property functions are thus expected to deal with topology independent aspects of element constant building for a given family of elements.

Thus the element integinfo call usually just transmits arguments to a property function that does most of the work. That means defining the contents of integ and constit columns. For example for an acoustic fluid, constit colums generated by p_solid BuildConstit contain $\left[\frac{1}{\rho C^2} \quad \eta \quad \frac{1}{\rho}\right]$.

Generic elements (hexa8, q4p, ...) all call p_solid BuildConstit. Depending on the property type coded in column 2 of the current material, p_solid attempts to call the associated m_
ti mat function with a BuildConstit command. If that fails, an attempt to call p_
ti mat is made (this allows to define a new familly of elements trough a single p_fcn
p_heat is such an example).

integ nominally contains MatId,ProId,NDofPerElt,NNodePerElt,IntegRuleNumber.

Const

Similarly, element constant generation of elements that support variable integration rules is performed for an element family. For example, p_solid const supports for 3D elastic solids, for 2D elastic solids and 3D acoustic fluid volumes. p_heat supports 2D and 3D element constant building for the heat equation.

Generic elements (hexa8, q4p, ...) all use the call [EltConst,NDNDim] = p_solid('Const',ElemF
integ, constit). User extendibility requires that the user be able to bypass the normal operation of p_solid const. This can be achieved by setting constit(1)=-1
and coding a property type in the second value (for example constit(1)=fe_mat('p_heat','SI',
The proper function is then called witht the same arguments as p_solid.

***_fcn** Expected commands common to both p_* and m_* functions are the following

Subtype

With no argument returns a cell array of strings associated with each subtype (maximum is 9). With a string input, it returns the numeric value of the subtype. With a numeric input, returns the string value of the subtype. See m_elastic for the reference implementation.

database

Returns a structure with reference materials or properties of this type. Additional strings can be used to give the user more freedom to build properties.

dbval

Mostly the same as database but replaces or appends rows in model.il (for element properties) or model.pl (for material properties).

PropertyUnitType

i1=p_function('PropertyUnitType',SubType) returns for each subtype the units of each value in the property row (column of pl).

This mechanism is used to automate unit conversions in Convert.

[list,repeat]=p_function('PropertyUnitTypeCell',SubType) returns a cell array describing the content of each column, the units and possibly a longer description of the variable. When properties can be repeated a variable number of times, use the repeat (example in p_shell for composites). This mechanism is used to generate graphical editors for properties.

Cell arrays describing each subtype give

- a label. This should be always the same to allow name based manipulations and should not contain any character that cannot be used in field names.
- a conversion value. Lists of units are given using fe_mat('convertSITM'). If the unit is within that list, the conversion value is the row number. If the unit is the ratio of two units in the list this is obtained using a non integer conversion value. Thus 9.004 corresponds to kg/m (9 is kg and 4 is m).
- a string describing the unit

5.14.5 Compiled element families in of_mk

of_mk is the C function used to handle all compiled element level compuations. Integration rules and shape derivatives are also supported as detailed in BuildNDN.

Generic multi-physic linear elements

This element family supports a fairly general definition of linear multi-physic elements whose element integration strategy is fully described by an EltConst data structure. hexa8 and p_solid serve as a prototype element function. Element matrix and load computations are implemented in the of_mk.c MatrixIntegration command with StrategyType=1, stress computations in the of_mk.c StressObserve command.

```
EltConst=hexa8('constants',[],[1 1 24 8],[]);
integrules('texstrain',EltConst)
EltConst=integrules('stressrule',EltConst);
integrules('texstress',EltConst)
```

Elements of this family are standard element functions (see section 5.14) and the element functions must thus return `node`, `prop`, `dof`, `line`, `patch`, `edge`, `face`, and `parent` values. The specificity is that all information needed to integrate the element is stored in an `EltConst` data structure that is initialized during the `fe_mknl GroupInit` phase.

For DOF definitions, the family uses an internal DOF sort where each field is given at all nodes sequentially $1x2x...8x1y...8y...$ while the more classical sort by node $1x1y...2x...$ is still used for external access (internal and external DOF sorting are discussed in section 5.14.7).

Each linear element matrix type is represented in the form of a sum over a set of integration points

$$k^{(e)} = \sum_{ji,jj} \sum_{jw} \left[\{B_{ji}\} D_{ji\ jk}(w(jw)) \{B_{jj}\}^T \right] J(w(jw)) W((jw)) \tag{5.3}$$

where the jacobian of the transformation from physical xyz to element rst coordinates is stored in `EltConst.jdet(jw)` and the weighting associated with the integration rule is stored in `EltConst.w(jw,4)`.

The relation between the `Case.GroupInfo constit` columns and the D_{ij} constitutive law matrix is defined by the cell array `EltConst.ConstitTopology` entries. For example, the strain energy of a acoustic pressure formulation (`p_solid ConstFluid`) is given by

```
constit(:,j1)=[1/rho/C2; eta ; 1/rho]
EltConst.MatrixTopology{1} = [3 0 0
                              0 3 0
                              0 0 3]
```

$$D = \begin{bmatrix} 1/\rho & 0 & 0 \\ 0 & 1/\rho & 0 \\ 0 & 0 & 1/\rho \end{bmatrix}$$

The integration rule for a given element is thus characterized by the strain observation matrix $B_{ji}(r,s,t)$ which relates a given strain component ϵ_{ji} and the nodal displacements. The generic linear element family assumes that the generalized strain components are linear functions of the shape functions and their derivatives in euclidian coordinates (xyz rather than rst).

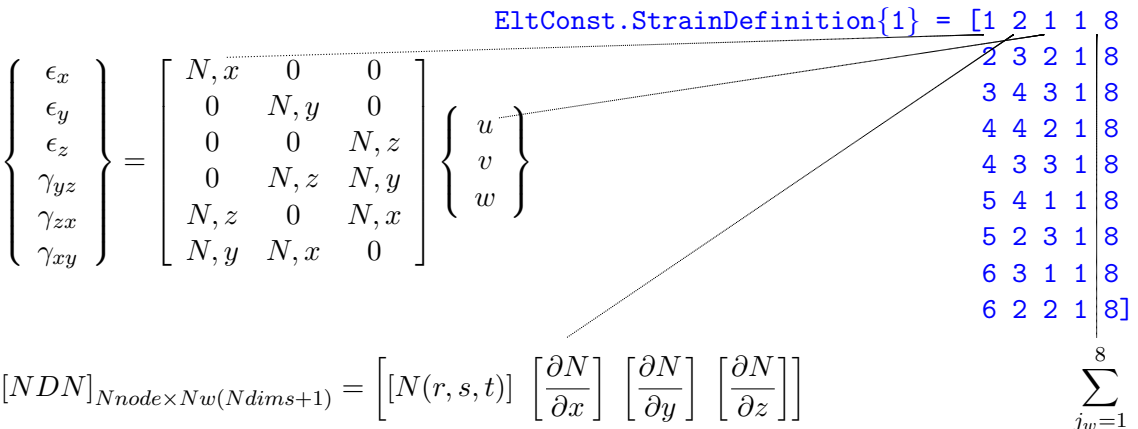
The first step of the element matrix evaluation is the evaluation of the `EltConst.NDN` matrix whose first Nw columns store shape functions, Nw next their derivatives with respect to x, then y and z for 3D elements

$$[NDN]_{Nnode \times Nw(Ndims+1)} = \left[[N(r,s,t)] \left[\frac{\partial N}{\partial x}\right] \left[\frac{\partial N}{\partial y}\right] \left[\frac{\partial N}{\partial z}\right] \right] \tag{5.4}$$

To improve speed the `EltConst.NDN` and associated `EltConst.jdet` fields are preallocated and reused for the assembly of element groups.

For each strain vector type, one defines an `int32` matrix

`EltConst.StrainDefinition{jType}` with each row describing `row, NDNBloc, DOF, NwStart, NwTot` giving the strain component number (these can be repeated since a given strain component can combine more than one field), the block column in NDN (block 1 is N, 4 is $\partial N/\partial z$), the field number, and the starting integration point associated with this strain component and the number of integration points needed to assemble the matrix. The default for `NwStart NwTot` is `1, Nw` but this formalism allows for differentiation of the integration strategies for various fields. The figure below illustrates this construction for classical mechanical strains.



$$[NDN]_{Nnode\times Nw(Ndims+1)} = \left[[N(r,s,t)] \left[\frac{\partial N}{\partial x}\right] \left[\frac{\partial N}{\partial y}\right] \left[\frac{\partial N}{\partial z}\right]\right] \qquad \sum_{j_w=1}^{8}$$

To help you check the validity of a given rule, you should fill the

`EltConst.StrainLabels{jType}` and `EltConst.DofLabels` fields and use the `integrules( 'texstrain', EltConst)` command to generate a LATEX printout of the rule you just generated.

The `.StrainDefinition` and `.ConstitTopology` information is combined automatically in `integrules` to generate `.MatrixIntegration` (`integrules MatrixRule` command) and `.StressRule` fields (`integrules StressRule` command). These tables once filed properly allow an automated integration of the element level matrix and stress computations in OpenFEM.

Generic RHS computations

Right hand side (load) computations can either be performed once (fixed set of loads) through `fe_load` which deals with multiple loads, or during an iterative process where a single RHS is assembled by `fe_mknl` into the second column of the state argument `dc.def(:,2)` along with the matrices when requiring the stiffness with `MatDes=1` or `MatDes=5` (in the second case, the forces are assumed following if implemented).

There are many classical forms of RHS, one thus lists here forms that are implemented in `of_mk.c MatrixIntegration`. Computations of these rules, requires that the `EltConst.VectMap` field by defined. Each row of `EltConst.RhsDefinition` specifies the procedure to be used for integration.

Two main strategies are supported where the fields needed for the integration of loads are stored either as colums of `dc.def` (for fields that can defined on DOFs of the model) or as entries in `InfoAtNode` (`Case.GroupInfo{7}`). In the later case, each column of `InfoAtNode` specifies an input field specified at nodes of the model

or with a NodePos field at specific nodes. The new strategy is compatible with both continuous and discontinous fields at each node. THIS HAS BEEN REVISED FOR 2007a and is still unstable.

Initialization of InfoAtNode is performed with fe_mknl('Init -gstate') calls.

Currently the only accepted format for rows of EltConst.RhsDefinition is

101(1) InfoAtNode1(2) InStep(3) NDNOff1(4) FDof1(5) NDNCol(6) NormalComp(7) w1(8) nwStep(9)

Where InfoAtNode1 gives the first row index in storing the field to be integrated in InfoAtNode. InStep gives the index step (3 for a 3 dimensional vector field), NDNOff1 gives the block offset in the NDN matrix (zero for the nominal shape function). FDof1 gives the offset in force DOFs for the current integration. NDNCol. If larger than -1, the normal component NormalComp designs a row number in EltConst.bas, which is used as a weighting coefficient. tt w1 gives the index of the first gauss point to be used (in C order starting at 0). nwStep gives the number of gauss points in the rule being used.

- volume forces not proportional to density

$$\int_{\Omega_0} f_v(x).du(x) = \{F_v\}_k = \sum_{j_w} \left(\{N_k(j_w)\}\{N_j(j_w)\} f_v(x_j)\right) J(j_w) W(j_w) \tag{5.5}$$

  are thus described by

```
opt.RhsDefinition=int32( ...
  [101 0 3 0     0 0 -1    rule+[-1 0];
   101 1 3 0     1 0 -1    rule+[-1 0];
   101 2 3 0     2 0 -1    rule+[-1 0]]);
```

  for 3D solids (see p_solid).

  Similarly, normal pressure is integrated as 3 volume forces over 3D surface elements with normal component weighting

$$\begin{array}{rcl} F_m & = & \int_{\partial\Omega_0} p(x) n_m(x).dv(x) \\ & = & \sum_{j_w} \left(\{N_k(j_w)\}\{N_j(j_w)\} p(x_j) n_m\right) J(j_w) W(j_w) \end{array} \tag{5.6}$$

- inertia forces (volume forces proportional to density)

$$F = \int_{\Omega_0} \rho(x) f_v(x).dv(x) \tag{5.7}$$

- stress forces (will be documented later)

Elastic3DNL fully anisotropic elastic elements in geometrically non-linear mechanics problems. Element matrix are implemented in the of_mk.c MatrixIntegration command with StrategyType=2 for the linear tangent matrix (MatType=5). Other computations are performed using generic elements (section 5.14.5) (mass MatType=2). This formulation family has been tested for the prediction of vibration responses under static pre-load.

Stress post-processing is implemented using the underlying linear element.

Simultaneous element matrix and right hand side computations are implemented in the of_mk.c MatrixIntegration command with StrategyType=3 for the linear tangent matrix (MatType=5). In this case (and only this case!!), the EltConst.NDN matrix is built as follow:
for $1 \leq jw \leq Nw$

$$[NDN]_{(Ndims+1)\times Nnode(Nw)} = \left[[NDN]^{jw}\right] \tag{5.8}$$

with

$$[NDN]^{jw}_{(Ndims+1)\times Nnode} = \begin{bmatrix} [N(r,s,t)]_{jw} \\ \left[\frac{\partial N}{\partial x}\right]_{jw} \\ \left[\frac{\partial N}{\partial y}\right]_{jw} \\ \left[\frac{\partial N}{\partial z}\right]_{jw} \end{bmatrix} \tag{5.9}$$

This implementation corresponds to case 31 of NDNSwitch function in of_mk_pre.c. The purpose is to use C-BLAS functions in element matrix and right hand side computations implemented in the same file (function Mecha3DintegH) to improve speed.

Other computations are performed using generic elements (section 5.14.5) (mass MatType=2). This formulation family has been tested for the RivlinCube test.

Stress post-processing is not implemented.

5.14.6 Non-linear iterations, what is done in of_mk

Non linear problems are characterized by the need to perform iterations with multiple assemblies of matrices and right hand sides (RHS). To optimize the performance, the nominal strategy for non-linear operations is to

- perform an initialization (standard of_mknl init call)
- define a deformation data structure dc with two columns giving respectively the current state and the non linear RHS.

At a given iteration, one resets the RHS and performs a single fe_mknl call that returns the current non-linear matrix and replaces the RHS by its current value

(note that fe_mknl actually modifies the input argument dc which is not an normal MATLAB behaviour but is needed here for performance)

```
% at init allocate DC structure
dc=struct('DOF',model.DOF,'def',zeros(length(model.DOF),2);
% ... some NL iteration mechanism here
dc.def(:,2)=0; % reset RHS at each iteration
k=fe_mknl('assemble not',model,Case,dc,5); % assemble K and RHS
```

Most of the work for generic elements is done within the of_mk MatrixIntegration command that is called by fe_mknl. Each call to the command performs matrix and RHS assembly for a full group of elements. Three strategies are currently implemented

- Linear multiphysic elements of arbitrary forms, see section 5.14.5
- Elastic3DNL general elastic elements for large, see section 5.14.5 transformation,
- Hyperelastic elements for large transformation problems. see section 5.14.5. These elements have been tested through the RivlinCube example.

5.14.7 Element function command reference

Nominally you should write topology independent element families, if hard coding is needed you can however develop new element functions.

In Matlab version, a typical element function is an .m or .mex file that is in your MATLAB path. In Scilab version, a typical element function is an .sci or mex file that is loaded into Scilab memory (see getf in Scilab on-line help).

The name of the function/file corresponds to the name of the element (thus the element bar1 is implemented through the bar1.m file)

General element information

To build a new element take q4p.m or q4p.sci as an example.

As for all Matlab or Scilab functions, the header is composed of a function syntax declaration and a help section. The following example is written for Matlab. For Scilab version, don't forget to replace % by //. In this example, the name of the created element is elem0.

For element functions the nominal format is

```
function [out,out1,out2]=elem0(CAM,varargin);
%elem0 help section
```

The element function should then contain a section for standard calls which let other functions know how the element behaves.

```
if isstr(CAM) %standard calls with a string command

 [CAM,Cam]=comstr(CAM,1); % remove blanks
 if comstr(Cam,'integinfo')
  % some code needed here
  out= constit; % real parameter describing the constitutive law
  out1=integ;   % integer (int32) parameters for the element
  out2=elmap;

 elseif comstr(Cam,'matcall')
  out=elem0('call');
  out1=1; % SymFlag
 elseif comstr(Cam,'call');     out = ['AssemblyCall'];
 elseif comstr(Cam,'rhscall');  out = ['RightHandSideCall'];
 elseif  comstr(Cam,'scall');   out = ['StressComputationCall'];
 elseif  comstr(Cam,'node');    out = [NodeIndices];
 elseif  comstr(Cam,'prop');    out = [PropertyIndices];
 elseif  comstr(Cam,'dof');     out = [ GenericDOF ];
 elseif comstr(Cam,'patch');
                        out = [ GenericPatchMatrixForPlotting ];
 elseif comstr(Cam,'edge');     out = [ GenericEdgeMatrix ];
 elseif comstr(Cam,'face');     out = [ GenericFaceMatrix ];
 elseif comstr(Cam,'sci_face'); out = [ SciFaceMatrix ];
 elseif comstr(Cam,'parent');   out = ['ParentName'];
 elseif comstr(Cam,'test')
    % typically one will place here a series of basic tests
 end
 return
end % of standard calls with string command
```

The expected outputs to these calls are detailed below.

`call,matcall`

Format string for element matrix computation call. Element functions must be able to give `fe_mk` the proper format to call them (note that superelements take precedence over element functions with the same name, so avoid calling a superelement `beam1`, etc.).

`matcall` is similar to call but used by `fe_mknl`. Some elements directly call the `of_mk` mex function thus avoiding significant loss of time in the element function. If your element is not directly supported by `fe_mknl` use `matcall=elem0('call')`.

The format of the call is left to the user and determined by `fe_mk` by executing the command `eCall=elem0('call')`. The default for the string `eCall` should be (see any of the existing element functions for an example)

```
  [k1,m1]=elem0(nodeE,elt(cEGI(jElt),:),...
```

```
pointers(:,jElt),integ,constit,elmap);
```

To define other proper calling formats, you need to use the names of a number of variables that are internal to `fe_mk`. `fe_mk` variables used as *output arguments of element functions* are

`k1` element matrix (must always be returned, for `opt(1)==0` it should be the stiffness, otherwise it is expected to be the type of matrix given by `opt(1)`)

`m1` element mass matrix (optional, returned for `opt(1)==0`, see below)

`[ElemF,opt,ElemP]=feutil('getelemf',elt(EGroup(jGroup),:),jGroup)` returns, for a given header row, the element function name `ElemF`, options `opt`, and parent name `ElemP`.

`fe_mk` and `fe_mknl` variables that can be used as *input arguments to element function* are listed in section 5.14.1.

`dof, dofcall`

Generic DOF definition vector. For user defined elements, the vector returned by `elem0('dof')` follows the usual DOF definition vector format (`NodeId.DofId` or `-1.DofId`) but is generic in the sense that node numbers indicate positions in the element row (rather than actual node numbers) and `-1` replaces the element identifier (if applicable).

For example the `bar1` element uses the 3 translations at 2 nodes whose number are given in position 1 and 2 of the element row. The generic DOF definition vector is thus `[1.01;1.02;1.03;2.01;2.01;2.03]`.

A `dofcall` command may be defined to bypass generic `dof` calls. In particular, this is used to implement elements where the number of DOFs depends on the element properties. The command should always return `out=elem0('dofcall');`. The actual DOF building call is performed in `p_solid('BuildDof')` which will call user `p_*.m` functions if needed.

Elements may use different DOF sorting for their internal computations.

`edge,face,patch,line,sci_face`

`face` is a matrix where each row describes the positions in the element row of nodes of the oriented face of a volume (conventions for the orientation are described under `integrules`). If some faces have fewer nodes, the last node should be repeated as needed. `feutil` can consider face sets with orientation conventions from other software.

`edge` is a matrix where each row describes the node positions of the oriented edge of a volume or a surface. If some edges have fewer nodes, the last node should be repeated as needed.

`line` (obsolete) is a vector describes the way the element will be displayed in the line mode (wire frame). The vector is generic in the sense that node numbers represent positions in the element row rather than actual node numbers. Zeros can be used to create a discontinuous line. `line` is now typically generated using information provided by `patch`.

`patch`. In MATLAB version, surface representations of elements are based on the use of MATLAB `patch` objects. Each row of the generic patch matrix gives the indices nodes. These are generic in the sense that node numbers represent positions in the element row rather than actual node numbers.

For example the `tetra4` solid element has four nodes in positions `1:4`. Its generic patch matrix is `[1 2 3;2 3 4;3 4 1;4 1 2]`. Note that you should not skip nodes but simply repeat some of them if various faces have different node counts.

`sci_face` is the equivalent of `patch` for use in the SCILAB implementation of *OpenFEM*. The difference between `patch` and `sci_face` is that, in SCILAB, a face must be described with 3 or 4 nodes. That means that, for a two nodes element, the last node must be repeated (in generallity, `sci_face = [1 2 2];`). For a more than 4 nodes per face element, faces must be cut in subfaces. The most important thing is to not create new nodes by the cutting of a face and to use all nodes. For example, 9 nodes quadrilateral can be cut as follows :

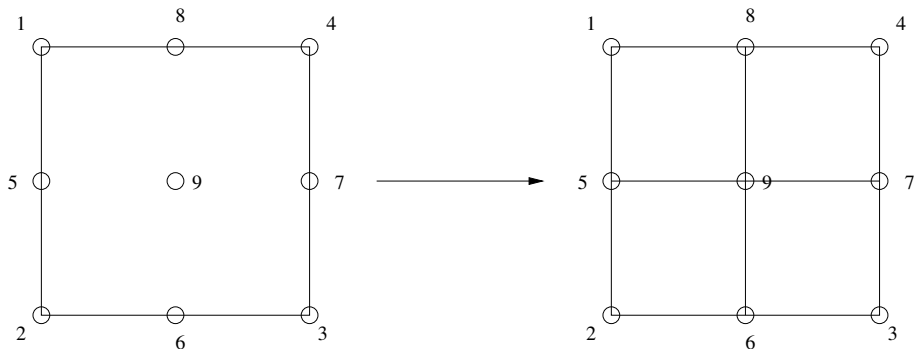


Figure 5.1: Lower order patch representation of a 9 node quadrilateral

but a 8 nodes quadrilaterals cannot by cut by this way. It can be cut as follows:

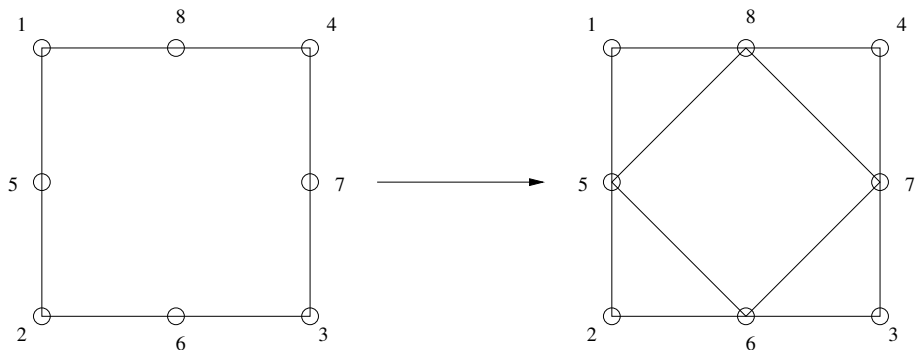


Figure 5.2: Lower order patch representation of a 8 node quadrilateral

`integinfo`, `BuildConstit`

Commands to resolve constants in elements and `p_function` respectively.

`[constit,integ,elmap]=elem0('integinfo',[MatId ProId],pl,il,model,Case)` is supposed to search `pl` and `il` for rows corresponding to `MatId` and `ProId` and return a real vector `constit` describing the element consitutive law and an integer vector `integ`.

`ElMap` is used to build the full matrix of an element which initially only gives it lower or upper triangular part. If a structure is return, `fe_mknl` can do some group wise processing (typically initialization of internal states).

In most elements, one uses `[constit,integ,elmap]=p_solid('buildconstit', [varargin{1};Ndof;Nnode],varargin{2:end})` since `p_solid` passes calls to other element property functions when needed.

`elmap` can also be used to pass structures and callbacks back to `fe_mknl`.

`node`

Vector of indices giving the position of nodes numbers in the element row. In general this vector should be `[1:n]` where `n` is the number of nodes used by the element.

`prop`

Vector of indices giving the position of `MatId`, `ProId` and `EltId` in the element row. In general this vector should be `n+[1 2 3]` where `n` is the number of nodes used by the element. If the element does not use any of these identifiers the index value should be zero (but this is poor practice).

`parent`

Parent element name. If your element is similar to a standard element (`beam1`, `tria3`, `quad4`, `hexa8`, etc.), declaring a parent allows the inheritance of properties. In particular you will be able to use functions, such as `fe_load` or parts of `femesh`, which only recognize standard elements.

`rhscall`

`rhscall` is a string that will be evaluated by `fe_load` when computing right hand side loads (volume and surface loads). Like `call` or `matcall`, the format of the call is determined by `fe_load` by executing the command `eCall=elem0('call')`. The default for the string `eCall` should be :

```
be=elem0(nodeE,elt(cEGI(jElt),:),pointers(:,jElt),...
                        integ,constit,elmap,estate);
```

The output argument `be` is the right hand side load. The inputs arguments are the same as those for `matcall` and `call`.

Matrix, load and stress computations

The calls with one input are followed by a section on element matrix assembly. For these calls the element function is expected to return an element DOF definition vector `idof` and an element matrix `k`. The type of this matrix is given in `opt(1)`. If `opt(1)==0`, both a stiffness `k` and a mass matrix `m` should be returned. See the `fe_mk MatType` section for a current list.

Take a look at `bar1` which is a very simple example of element function.

A typical element assembly section is as follows :

```
% elem0 matrix assembly section

% figure out what the input arguments are
node=CAM;   elt=varargin{1};
point=varargin{2};  integ=varargin{3};
constit=varargin{4}; elmap=varargin{5};
typ=point(5);

% outputs are [k,m] for opt(1)==0
%             [mat] for other opt(1)
switch point(5)
case 0
 [out,out1] = ... % place stiffness in out and mass in out1
case 1
  out= ...  % compute stiffness
case 2
  out= ...  % compute mass
case 100
  out= ...  % compute right hand side
```

```
case 200
  out= ...  % compute stress  ...
otherwise
  error('Not a supported matrix type');
end
```

Distributed load computations (surface and volume) are handled by `fe_load`. Stress computations are handled by `fe_stres`.

There is currently no automated mechanism to allow users to integrate such computations for their own elements without modifying `fe_load` and `fe_stres`, but this will appear later since it is an obvious maintenance requirement.

The mechanism that will be used will be similar to that used for matrix assembly. The element function will be required to provide calling formats when called with `elem0('fsurf')` for surface loads, `elem0('fvol')` for volume loads, and `elem0('stress')` for stresses. `fe_load` and `fe_stres` will then evaluate thes calls for each element.

5.15 Variable names and progamming rules

The following rules are used in programming OpenFEM as is makes reading the source code easier.

`carg`	index of current argument. For functions with variable number of inputs, one seeks the next argument with `NewArg=varargincarg;carg=carg+1;`
`CAM,Cam`	string command to be interpreted. `Cam` is the lower case version of `CAM`.
`j1,j2,j3` ...	loop indices.
`jGroup,jElt,jW`	indices for element groups, elements, integration points
`i,j`	unit imaginary $\sqrt{-1}$. `i,j` should never be used as indices to avoid any problem overloading their default value.
`i1,i2,i3` ...	integer values intermediate variables
`r1,r2,r3` ...	real valued variables or structures
`ind,in2,in3` ...	vectors of indices, `cind` is used to store the complement of `ind` when applicable.
`out,out1,out2` ...	output variables

The following names are also used throughout the toolbox functions

`node,FEnode`	nodes
`NNode`	reindexing vector verifies `NodeInd=NNode(NodeId)`. Can be built using `NNode=sparse(node(:,1),1,1:size(node,1))`.

5.16 Legacy information

This section gives data that is no longer used but is important enough not to be deleted.

5.16.1 Legacy 2D elements

These elements support isotropic and 2-D anisotropic materials declared with a material entry described in `m_elastic`. Element property declarations are `p_solid` subtype 2 entries

```
[ProId  fe_mat('p_solid','SI',2)  f  N  0]
```

Where

`f`	Formulation : `0` plane stress, `1` plane strain, `2` axisymmetric.
`N`	Fourier coefficient for axisymmetric formulations
`Integ`	set to zero to select this family of elements.

The xy plane is used with displacement DOFs `.01` and `.02` given at each node. Element matrix calls are implemented using `.c` files called by `of_mk_subs.c` and handled by the element function itself, while load computations are handled by `fe_load`. For integration rules, see section 5.16.2. The following elements are supported

- `q4p (plane stress/strain)` uses the `et*2q1d` routines for plane stress and plane strain.
- `q4p (axisymmetric)` uses the `et*aq1d` routines for axisymmetry. The radial u_r and axial u_z displacement are bilinear functions over the element.
- `q5p (plane stress/strain)` uses the `et*5noe` routines for axisymmetry.

  There are five nodes for this incompressible quadrilateral element, four nodes at the vertices and one at the intersection of the two diagonals.
- `q8p` uses the `et*2q2c` routines for plane stress and plane strain and `et*aq2c` for axisymmetry.
- `q9a` is a plane axisymmetric element with Fourier support. It uses the `e*aq2c` routines to generate matrices.
- `t3p` uses the `et*2p1d` routines for plane stress and plane strain and `et*ap1d` routines for axisymmetry.

  The displacement (u,v) are assumed to be linear functions of (x,y) (*Linear Triangular Element*), thus the strain are constant (*Constant Strain Triangle*).
- `t6p` uses the `et*2p2c` routines for plane stress and plane strain and `et*ap2c` routines for axisymmetry.

5.16.2 Rules for elements in `of_mk_subs`

hexa8, hexa20

The hexa8 and hexa20 elements are the standard 8 node 24 DOF and 20 node 60 DOF brick elements.

The hexa8 element uses the et*3q1d routines.

hexa8 volumes are integrated at 8 Gauss points

$\omega_i = \frac{1}{8}$ for $i = 1, 4$

b_i for $i = 1, 4$ as below, with $z = \alpha_1$

b_i for $i = 4, 8$ as below, with $z = \alpha_2$

hexa8 surfaces are integrated using a 4 point rule

$\omega_i = \frac{1}{4}$ for $i = 1, 4$

$b_1 = (\alpha_1, \alpha_1)$, $b_2 = (\alpha_2, \alpha_1)$, $b_3 = (\alpha_2, \alpha_2)$ and $b_4 = (\alpha_1, \alpha_2)$

with $\alpha_1 = \frac{1}{2} - \frac{1}{2\sqrt{3}} = 0.2113249$ and $\alpha_2 = \frac{1}{2} + \frac{1}{2\sqrt{3}} = 0.7886751$.

The hexa20 element uses the et*3q2c routines.

hexa20 volumes are integrated at 27 Gauss points $\omega_l = w_i w_j w_k$ for $i, j, k = 1, 3$

with

$w_1 = w_3 = \frac{5}{18}$ and $w_2 = \frac{8}{18}$ $b_l = (\alpha_i, \alpha_j, \alpha_k)$ for $i, j, k = 1, 3$

with

$\alpha_1 = \frac{1-\sqrt{\frac{3}{5}}}{2}$, $\alpha_2 = 0.5$ and $\alpha_3 = \frac{1+\sqrt{\frac{3}{5}}}{2}$

$\alpha_1 = \frac{1-\sqrt{\frac{3}{5}}}{2}$, $\alpha_2 = 0.5$ and

hexa20 surfaces are integrated at 9 Gauss points $\omega_k = w_i w_j$ for $i, j = 1, 3$ with

w_i as above and $b_k = (\alpha_i, \alpha_j)$ for $i, j = 1, 3$

with $\alpha_1 = \frac{1-\sqrt{\frac{3}{5}}}{2}$, $\alpha_2 = 0.5$ and $\alpha_3 = \frac{1+\sqrt{\frac{3}{5}}}{2}$.

penta6, penta15

The penta6 and penta15 elements are the standard 6 node 18 DOF and 15 node 45 DOF pentahedral elements. A derivation of these elements can be found in [2].

The penta6 element uses the et*3r1d routines.

penta6 volumes are integrated at 6 Gauss points

Points b_k	x	y	z
1	a	a	c
2	b	a	c
3	a	b	c
4	a	a	d
5	b	a	d
6	a	b	d

with $a = \frac{1}{6} = .16667$, $b = \frac{4}{6} = .66667$, $c = \frac{1}{2} - \frac{1}{2\sqrt{3}} = .21132$, $d = \frac{1}{2} + \frac{1}{2\sqrt{3}} = .78868$

`penta6` surfaces are integrated at 3 Gauss points for a triangular face (see `tetra4`) and 4 Gauss points for a quadrangular face (see `hexa8`).

`penta15` volumes are integrated at 21 Gauss points with the 21 points formula

$a = \frac{9-2\sqrt{15}}{21}$, $b = \frac{9+2\sqrt{15}}{21}$,

$c = \frac{6+\sqrt{15}}{21}$, $d = \frac{6-\sqrt{15}}{21}$,

$e = 0.5(1 - \sqrt{\frac{3}{5}})$,

$f = 0.5$ and $g = 0.5(1 + \sqrt{\frac{3}{5}})$

$\alpha = \frac{155-\sqrt{15}}{2400}$, $\beta = \frac{5}{18}$,

$\gamma = \frac{155+\sqrt{15}}{2400}$, $\delta = \frac{9}{80}$ and $\epsilon = \frac{8}{18}$.

Positions and weights of the 21 Gauss point are

Points b_k	x	y	z	weight ω_k
1	d	d	e	$\alpha.\beta$
2	b	d	e	$\alpha.\beta$
3	d	b	e	$\alpha.\beta$
4	c	a	e	$\gamma.\beta$
5	c	c	e	$\gamma.\beta$
6	a	c	e	$\gamma.\beta$
7	$\frac{1}{3}$	$\frac{1}{3}$	e	$\delta.\beta$
8	d	d	f	$\alpha.\epsilon$
9	b	d	f	$\alpha.\epsilon$
10	d	b	f	$\alpha.\epsilon$
11	c	a	f	$\gamma.\epsilon$
12	c	c	f	$\gamma.\epsilon$
13	a	c	f	$\gamma.\epsilon$
14	$\frac{1}{3}$	$\frac{1}{3}$	f	$\delta.\epsilon$
15	d	d	g	$\alpha.\beta$
16	b	d	g	$\alpha.\beta$
17	d	b	g	$\alpha.\beta$
18	c	a	g	$\gamma.\beta$
19	c	c	g	$\gamma.\beta$
20	a	c	g	$\gamma.\beta$
21	$\frac{1}{3}$	$\frac{1}{3}$	g	$\delta.\beta$

`penta15` surfaces are integrated at 7 Gauss points for a triangular face (see `tetra10`) and 9 Gauss points for a quadrangular face (see `hexa20`).

`tetra4, tetra10`

The `tetra4` element is the standard 4 node 12 DOF trilinear isoparametric solid element. `tetra10` is the corresponding second order element.

You should be aware that this element can perform very badly (for poor aspect ratio, particular loading conditions, etc.) and that higher order elements should be used instead.

The `tetra4` element uses the `et*3p1d` routines.

`tetra4` volumes are integrated at the 4 vertices $\omega_i = \frac{1}{4}$ for $i = 1, 4$ and $b_i = S_i$ the i-th element vertex.

`tetra4` surfaces are integrated at the 3 vertices with $\omega_i = \frac{1}{3}$ for $i = 1, 3$ and $b_i = S_i$ the i-th vertex of the actual face

The `tetra10` element is second order and uses the `et*3p2c` routines.

`tetra10` volumes are integrated at 15 Gauss points

Points b_k	λ_1	λ_2	λ_3	λ_4	weight ω_k
1	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{8}{405}$
2	b	a	a	a	α
3	a	b	a	a	α
4	a	a	b	a	α
5	a	a	a	b	α
6	d	c	c	c	β
7	c	d	c	c	β
8	c	c	d	c	β
9	c	c	c	d	β
10	e	e	f	f	γ
11	f	e	e	f	γ
12	f	f	e	e	γ
13	e	f	f	e	γ
14	e	f	e	f	γ
15	f	e	f	e	γ

with $a = \frac{7-\sqrt{15}}{34} = 0.0919711$, $b = \frac{13+3\sqrt{15}}{34} = 0.7240868$, $c = \frac{7+\sqrt{15}}{34} = 0.3197936$, $d = \frac{13-3\sqrt{15}}{34} = 0.0406191$, $e = \frac{10-2\sqrt{15}}{40} = 0.0563508$, $f = \frac{10+2\sqrt{15}}{40} = 0.4436492$

and $\alpha = \frac{2665+14\sqrt{15}}{226800}$, $\beta = \frac{2665-14\sqrt{15}}{226800}$ et $\gamma = \frac{5}{567}$

λ_j for $j = 1, 4$ are barycentric coefficients for each vertex S_j :

$b_k = \sum_{j=1,4} \lambda_j S_j$ for $k = 1, 15$

`tetra10` surfaces are integrated using a 7 point rule

Points b_k	λ_1	λ_2	λ_3	weight ω_k
1	c	d	c	α
2	d	c	c	α
3	c	c	d	α
4	b	b	a	β
5	a	b	b	β
6	b	a	b	β
7	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	γ

with $\gamma = \frac{9}{80} = 0.11250$, $\alpha = \frac{155-\sqrt{15}}{2400} = 0.06296959$, $\beta = \frac{155+\sqrt{15}}{2400} = 0.066197075$ and $a = \frac{9-2\sqrt{15}}{21} = 0.05961587$, $b = \frac{6+\sqrt{15}}{21} = 0.47014206$, $c = \frac{6-\sqrt{15}}{21} = 0.10128651$, $d = \frac{9+2\sqrt{15}}{21} = 0.797427$

λ_j for $j = 1, 3$ are barycentric coefficients for each surface vertex S_j :

$b_k = \sum_{j=1,3} \lambda_j S_j$ for $k = 1, 7$

`q4p` (plane stress/strain)

The displacement (u,v) are bilinear functions over the element.

For surfaces, q4p uses numerical integration at the corner nodes with $\omega_i = \frac{1}{4}$ and $b_i = S_i$ for $i = 1, 4$.

For edges, q4p uses numerical integration at each corner node with $\omega_i = \frac{1}{2}$ and $b_i = S_i$ for $i = 1, 2$.

q4p (axisymmetric)

For surfaces, q4p uses a 4 point rule with

- $\omega_i = \frac{1}{4}$ for $i = 1, 4$
- $b_1 = (\alpha_1, \alpha_1)$, $b_2 = (\alpha_2, \alpha_1)$, $b_3 = (\alpha_2, \alpha_2)$, $b_4 = (\alpha_1, \alpha_2)$ with $\alpha_1 = \frac{1}{2} - \frac{1}{2\sqrt{3}} = 0.2113249$ and $\alpha_2 = \frac{1}{2} + \frac{1}{2\sqrt{3}} = 0.7886751$

For edges, q4p uses a 2 point rule with

- $\omega_i = \frac{1}{2}$ for $i = 1, 2$
- $b_1 = \alpha_1$ and $b_2 = \alpha_2$ the 2 gauss points of the edge.

q5p (plane stress/strain)

For surfaces, q5p uses a 5 point rule with $b_i = S_i$ for $i = 1, 4$ the corner nodes and b_5 the node 5.

For edges, q5p uses a 1 point rule with $\omega = \frac{1}{2}$ and b the midside node.

q8p (plane stress/strain)

For surfaces, q8p uses a 9 point rule with

- $\omega_k = w_i w_j$ for $i, j = 1, 3$ with $w_1 = w_3 = \frac{5}{18}$ et $w_2 = \frac{8}{18}$
- $b_k = (\alpha_i, \alpha_j)$ for $i, j = 1, 3$ with $\alpha_1 = \frac{1-\sqrt{\frac{3}{5}}}{2}$, $\alpha_2 = 0.5$ and $\alpha_3 = \frac{1+\sqrt{\frac{3}{5}}}{2}$

For edges, q8p uses a 3 point rule with

- $\omega_1 = \omega_2 = \frac{1}{6}$ and $\omega_3 = \frac{4}{6}$
- $b_i = S_i$ for $i = 1, 2$ corner nodes of the edge et b_3 the midside.

q8p (axisymmetric)

For surfaces, q8p uses a 9 point rule with

- $\omega_k = w_i w_j$ for $i, j = 1, 3$ with $w_1 = w_3 = \frac{5}{18}$ and $w_2 = \frac{8}{18}$

- $b_k = (\alpha_i, \alpha_j)$ for $i, j = 1, 3$
  with $\alpha_1 = \frac{1-\sqrt{\frac{3}{5}}}{2}$, $\alpha_2 = 0.5$ and $\alpha_3 = \frac{1+\sqrt{\frac{3}{5}}}{2}$

For edges, q8p uses a 3 point rule with

- $\omega_1 = \omega_3 = \frac{5}{18}$, $\omega_2 = \frac{8}{18}$
- $b_1 = \frac{1-\sqrt{\frac{3}{5}}}{2} = 0.1127015$, $b_2 = 0.5$ and $b_3 = \frac{1+\sqrt{\frac{3}{5}}}{2} = 0.8872985$

t3p (plane stress/strain)

For surfaces, t3p uses a 3 point rule at the vertices with $\omega_i = \frac{1}{3}$ and $b_i = S_i$.

For edges, t3p uses a 2 point rule at the vertices with $\omega_i = \frac{1}{2}$ and $b_i = S_i$.

t3p (axisymmetric)

For surfaces, t3p uses a 1 point rule at the barycenter ($b_1 = G$) with $\omega_1 = \frac{1}{2}$.

For edges, t3p uses a 2 point rule at the vertices with $\omega_i = \frac{1}{2}$ and $b_1 = \frac{1}{2} - \frac{2}{2\sqrt{3}}$ and $b_2 = \frac{1}{2} + \frac{2}{2\sqrt{3}}$.

t6p (plane stress/strain)

For surfaces, t6p uses a 3 point rule with

- $\omega_i = \frac{1}{3}$ for $i = 1, 6$
- $b_i = S_{i+3,i+4}$ the three midside nodes.

For edges, t6p uses a 3 point rule

- $\omega_1 = \omega_2 = \frac{1}{6}$ and $\omega_3 = \frac{4}{6}$
- $b_i = S_i, i = 1, 2$ the i-th vertex of the actual edge and $b_3 = S_{i,i+1}$ the midside.

t6p (axisymmetric)

For surfaces, t6p uses a 7 point rule

Points b_k	λ_1	λ_2	λ_3	weight ω_k
1	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	a
2	α	β	β	b
3	β	β	α	b
4	β	α	β	b
5	γ	γ	δ	c
6	δ	γ	γ	c
7	γ	δ	γ	c

with :

$a = \frac{9}{80} = 0.11250$, $b = \frac{155+\sqrt{15}}{2400} = 0.066197075$ and
$c = \frac{155-\sqrt{15}}{2400} = 0.06296959$

$\alpha = \frac{9-2\sqrt{15}}{21} = 0.05961587$, $\beta = \frac{6+\sqrt{15}}{21} = 0.47014206$
$\gamma = \frac{6-\sqrt{15}}{21} = 0.10128651$, $\delta = \frac{9+2\sqrt{15}}{21} = 0.797427$

λ_j for $j = 1, 3$ are barycentric coefficients for each vertex S_j :

$b_k = \sum_{j=1,3} \lambda_j S_j$ for $k = 1, 7$

For edges, `t6p` uses a 3 point rule with $\omega_1 = \omega_3 = \frac{5}{18}$, $\omega_2 = \frac{8}{18}$

$b_1 = \frac{1-\sqrt{\frac{3}{5}}}{2} = 0.1127015$, $b_2 = 0.5$ and $b_3 = \frac{1+\sqrt{\frac{3}{5}}}{2} = 0.8872985$

6

Element reference

Element functions supported by *OpenFEM* are listed below. The rule is to have element families (2D and 3D) with families of formulations selected through element properties and implemented for all standard shapes

3-D volume element shapes	
hexa8	8-node 24-DOF brick
hexa20	20-node 60-DOF brick
hexa27	27-node 81-DOF brick
penta6	6-node 18-DOF pentahedron
penta15	15-node 45-DOF pentahedron
tetra4	4-node 12-DOF tetrahedron
tetra10	10-node 30-DOF tetrahedron

2-D volume element shapes	
q4p	4-node quadrangle
q5p	5-node quadrangle
q8p	8-node quadrangle
q9a	9-node quadrangle
t3p	3-node 6-DOF triangle
t6p	6-node 12-DOF triangle

Supported problem formulations are listed in section 3.2, in particular one considers 2D and 3D elasticity, acoustics, hyperelasticity, fluid/structure coupling, piezoelectric volumes, ...

Other elements, non generic elements, are listed below

3-D plate/shell Elements	
dktp	3-node 9-DOF discrete Kirchoff plate
mitc4	4-node 20-DOF shell
quadb	quadrilateral 4-node 20/24-DOF plate/shell
quad9	(display only)
quadb	quadrilateral 8-node 40/48-DOF plate/shell
tria3	3-node 15/18-DOF thin plate/shell element
tria6	(display only)

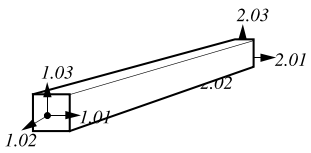
Other elements	
bar1	standard 2-node 6-DOF bar
beam1	standard 2-node 12-DOF Bernoulli-Euler beam
beam1t	pretensionned 2-node 12-DOF Bernoulli-Euler beam
beam3	(display only)
celas	scalar springs and penalized rigid links
mass1	concentrated mass/inertia element
mass2	concentrated mass/inertia element with offset
rigid	handling of linearized rigid links

Utility elements	
fe_super	generic element support
integrules	FEM integration rule support

bar1

Purpose Element function for a 6 DOF traction-compression bar element.

Description The `bar1` element corresponds to the standard linear interpolation for axial traction-compression. The element DOFs are the standard translations at the two end nodes (DOFs `.01` to `.03`).



In a model description matrix, *element property rows* for `bar1` elements follow the standard format (see section 5.14).

```
[n1 n2 MatID ProID EltID]
```

Isotropic elastic materials are the only supported (see `m_elastic`).

For supported element properties see `p_beam`. Currently, `bar1` only uses the element area `A` with the format

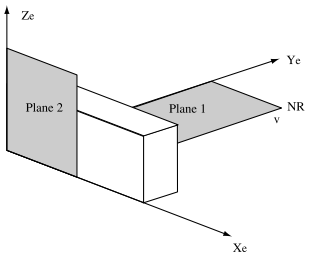
```
[ProID  Type   0  0  0 A]
```

See also `m_elastic`, `p_beam`, `fe_mk`, `feplot`

beam1, beam1t

Purpose Element function for a 12 DOF beam element. `beam1t` is a 2 node beam with pretension available for non-linear cable statics and dynamics.

Description `beam1`



In a model description matrix, *element property rows* for `beam1` elements follow the format

```
[n1 n2 MatID ProID nR 0 0 EltID p1 p2 x1 y1 z1 x2 y2 z2]
```

where

`n1,n2`	node numbers of the nodes connected
`MatID`	material property identification number
`ProID`	element section property identification number
`nr 0 0`	number of node not in the beam direction defining bending plane 1 in this case $\{v\}$ is the vector going from `n1` to `nr`. If `nr` is undefined it is assumed to be located at position [1.5 1.5 1.5].
`vx vy vz`	alternate method for defining the bending plane 1 by giving the components of a vector in the plane but not collinear to the beam axis. If `vy` and `vz` are zero, `vx` **must not be an integer**. `MAP=beam1t('map',model)` returns a normal vector MAP giving the vector used for bending plane 1. This can be used to check your model.
`p1,p2`	pin flags. These give a list of DOFs to be released (condensed before assembly). For example, 456 will release all rotation degrees of freedom. Note that the DOFS are defined in the local element coordinate system.
`x1,...`	optional components in global coordinate system of offset vector at node 1 (default is no offset)
`x2,...`	optional components of offset vector at node 2

Isotropic elastic materials are the only supported (see `m_elastic`). `p_beam` describes the section property format and associated formulations.

`beam1t`

This element has an internal state where each colum of `Case.GroupInfo{5}` gives the local basis, element length and tension `[bas(:);L;T]`.

This is a sample example how to impose a pre-tension :

beam1, beam1t

```
model=femesh('testbeam1 divide 10');
model=fe_case(model,'fixdof','clamp',[1;2;.04;.02;.01;.05]);
model.Elt=feutil('set group 1 name  beam1t',model);
[Case,model.DOF]=fe_mknl('init',model);
m=fe_mknl('assemble',model,Case,2);
k=fe_mknl('assemble',model,Case,1);
f1=fe_eig(m,k,[5 10]);
Case.GroupInfo{1,5}(11,:)=1.5e6; % tension
k1=fe_mknl('assemble',model,Case,1);
f1=[f1 fe_eig(m,k1,[5 10])]  % Note the evolution of frequencies
```

See also p_beam, m_elastic, fe_mk, feplot

celas,cbush

Purpose element function for scalar springs and penalized rigid links

Description In an model description matrix a group of `celas` elements starts with a header row `[Inf abs('celas') 0 ...]` followed by element property rows following the format `[n1 n2 DofID1 DofID2 ProID EltID Kv Mv Cv Bv]` with

`n1,n2` node numbers of the nodes connected. Grounded springs are obtained by setting `n1` or `n2` to `0`.

`DofID` Identification of selected DOFs.
For rigid links, the first node defines the rigid body motion. `DofID` (positive) defines which DOFs of the slave node are connected by the constraint. Thus `[1 2 123 0 0 0 1e14]` will only impose translations of node `2` are imposed by motion of node `1`, while `[1 2 123456 0 0 0 1e14]` will also penalize the difference in rotations.
For scalar springs, `DofID1` (negative) defines which DOFs of node 1 are connected to which of node 2. `DofID2` can be used to specify different DOFs on the 2 nodes. For example `[1 2 -123 231 0 0 1e14]` connects DOFs 1.01 to 2.02, etc.

`ProID` Optional property identification number (see format below)

`Kv` Optional stiffness value used as a weighting associated with the constraint. If `Kv` is zero (or not given), the default value in the element property declaration is used. If this is still zero, `Kv` is set to `1e14`.

`p_spring` properties for `celas` elements take the form `[ProID type KvDefault m c eta S]`

Below is the example of a 2D beam on elastic supports.

```
model=femesh('testbeam1 divide 10');
model=fe_case(model,'FixDof','2D',[.01;.02;.04]);
model.Elt(end+1,1:6)=[Inf abs('celas')]; % spring supports
model.Elt(end+[1:2],1:7)=[1 0 -13 0   0 0 1e5;2 0 -13 0   0 0 1e5];
def=fe_eig(model,[5 10 0]); feplot(model,def);
```

`cbush`

The element property row is defined by

`n1 n2 MatId ProId EltId x1 x2 x3 CID S OCID S1 S2 S3`

The orientation of the spring can be specified, by using distinct `n1,n2`, giving components `x1,x2,x3` of an orientation vector (x1 should not be integer if x2 and x3 are zero), a node number as `NodeIdRef,0,0`, the specification of a coordinate system `CID`.

The spring/damper is nominally located at the midpoint of n1,n2 (S=0.5). To use

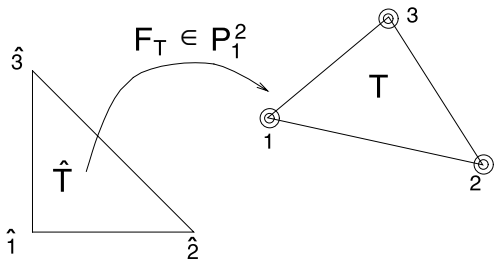
another location, specify a non-zero OCID and an offset `S1,S2,S3`.

See also `p_spring`, `rigid`

dktp

Purpose 2-D 9-DOF Discrete Kirchhoff triangle

Description



In a model description matrix, **element property rows** for `dktp` elements follow the standard format

```
[n1 n2 n3 MatID ProID EltID Theta]
```

giving the node identification numbers `ni`, material `MatID`, property `ProID`. Other **optional** information is `EltID` the element identifier, `Theta` the angle between material x axis and element x axis (currently unused)

The elements support isotropic materials declared with a material entry described in `m_elastic`. Element property declarations follow the format described in `p_shell`.

The `dktp` element uses the `et*dktp` routines.

There are three vertices nodes for this triangular Kirchhoff plate element and the normal deflection $W(x,y)$ is cubic along each edge.

We start with a 6-node triangular element with a total $D.O.F = 21$:

- five degrees of freedom at corner nodes :

$$W(x,y)\ ,\ \frac{\partial\, W}{\partial x}\ ,\ \frac{\partial\, W}{\partial y}\ ,\ \theta_x\ ,\ \theta_y\ \ (deflection\ W\ \ and\ \ rotations\ \ \theta)$$

- two degrees of freedom θ_x and θ_y at mid side nodes.

Then, we impose no transverse shear deformation $\gamma_{xz} = 0$ and $\gamma_{yz} = 0$ at selected nodes to reduce the total $DOF = 21 - 6 * 2 = 9$:

- three degrees of freedom at each of the vertices of the triangle.

$$W(x,y)\ ,\ \theta_x = (\frac{\partial\, W}{\partial x})\ ,\ \theta_y = (\frac{\partial\, W}{\partial y})$$

The coordinates of the reference element's vertices are $\hat{S}_1(0.,0.)$, $\hat{S}_2(1.,0.)$ and $\hat{S}_3(0.,1.)$.

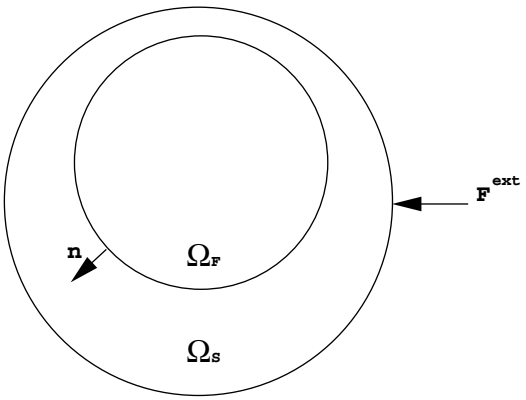
Surfaces are integrated using a 3 point rule $\omega_k = \frac{1}{3}$ and b_k mid side node.

See also `fe_mat`, `m_elastic`, `p_shell`, `fe_mk`, `feplot`

fsc

Purpose Fluid structure/coupling with non-linear follower pressure support.

Description Elasto-acoustic coupling is used to model structures containing a compressible, non-weighing fluid, with or without a free surface.



The FE formulation for this type of problem can be written as [**?**]

$$s^2 \begin{bmatrix} M & 0 \\ C^T & K_p \end{bmatrix} \begin{Bmatrix} q \\ p \end{Bmatrix} + \begin{bmatrix} K(s) & -C \\ 0 & F \end{bmatrix} \begin{Bmatrix} q \\ p \end{Bmatrix} = \begin{Bmatrix} F^{ext} \\ 0 \end{Bmatrix} \tag{6.1}$$

with q the displacements of the structure, p the pressure variations in the fluid and F^{ext} the external load applied to the structure, where

$$\begin{array}{c} \int_{\Omega_S} \sigma_{ij}(u)\epsilon_{ij}(\delta u)dx \Rightarrow \delta q^T K q \\ \int_{\Omega_S} \rho_S u.\delta u dx \Rightarrow \delta q^T M q \\ \frac{1}{\rho_F}\int_{\Omega_F} \nabla p \nabla \delta p dx \Rightarrow \delta p^T F p \\ \frac{1}{\rho_F c^2}\int_{\Omega_F} p\delta p dx \Rightarrow \delta p^T K_p p \\ \int_{\Sigma} p\delta u.n dx \Rightarrow \delta q^T C p \end{array} \tag{6.2}$$

Follower force One uses the identity

$$n\,dS = \frac{\partial \underline{x}}{\partial r} \wedge \frac{\partial \underline{x}}{\partial s}\,drds, \tag{6.3}$$

where (r, s) designate local coordinates of the face (assumed such that the normal is outgoing). Work of the pressure is thus:

$$\delta W_p = -\int_{r,s} \Pi\,(\frac{\partial \underline{x}}{\partial r} \wedge \frac{\partial \underline{x}}{\partial s}) \cdot \delta \underline{v}\,drds. \tag{6.4}$$

On thus must add the non-linear stiffness term:

$$-d\delta W_p = \int_{r,s} \Pi\,(\frac{\partial d\underline{u}}{\partial r} \wedge \frac{\partial \underline{x}}{\partial s} + \frac{\partial \underline{x}}{\partial r} \wedge \frac{\partial d\underline{u}}{\partial s}) \cdot \delta \underline{v}\,drds. \tag{6.5}$$

Using $\frac{\partial \underline{x}}{\partial r} = \{x_{1,r}\ x_{2,r}\ x_{3,r}\}^T$ (idem for s), and also

$$[Axr] = \begin{pmatrix} 0 & -x_{,r3} & x_{,r2} \\ x_{,r3} & 0 & -x_{,r1} \\ -x_{,r2} & x_{,r1} & 0 \end{pmatrix}, \quad [Axs] = \begin{pmatrix} 0 & -x_{,s3} & x_{,s2} \\ x_{,s3} & 0 & -x_{,s1} \\ -x_{,s2} & x_{,s1} & 0 \end{pmatrix},$$

this results in

$$(\frac{\partial d\underline{x}}{\partial r} \wedge \frac{\partial \underline{x}}{\partial s} + \frac{\partial \underline{x}}{\partial r} \wedge \frac{\partial d\underline{x}}{\partial s}) \cdot \delta \underline{v} = \{\delta q_{ik}\}^T \{N_k\} (Axr_{ij}\{N_{l,s}\}^T - Axs_{ij}\{N_{l,r}\}^T)\{dq_j\}. \tag{6.6}$$

Tests : `fsc3 testsimple` and `fsc3 test`.

In the RivlinCube test (see section 4.1), the pressure on each free face is given by

$$\begin{aligned} \Pi_1 &= -\tfrac{1+\lambda_1}{(1+\lambda_2)(1+\lambda_3)}\Sigma_{11} \quad on \quad face \quad (x_1 = l_1) \\ \Pi_2 &= -\tfrac{1+\lambda_2}{(1+\lambda_1)(1+\lambda_3)}\Sigma_{22} \quad on \quad face \quad (x_2 = l_2) \\ \Pi_3 &= -\tfrac{1+\lambda_3}{(1+\lambda_1)(1+\lambda_2)}\Sigma_{33} \quad on \quad face \quad (x_3 = l_3). \end{aligned}$$

See also `flui4`, `m_elastic`

hexa8, penta6, tetra4, and other 3D volumes

Purpose Topology holders for 3D volume elements.

Description The `hexa8` `hexa20` `hexa27`, `penta6` `penta15` `tetra4` and `tetra10` elements are standard topology reference for 3D volume FEM problems.

In a model description matrix, **element property rows** for `hexa8` and `hexa20` elements follow the standard format with no element property used. The generic format for an element containing i nodes is `[n1 ... ni MatID ProId EltId]`. For example, the `hexa8` format is `[n1 n2 n3 n4 n5 n6 n7 n8 MatID ProId EltId]`.

These elements only define topologies, the nature of the problem to be solved should be specified using a property entry, see section 3.2 for supported problems and `p_solid`, `p_heat`, ... for formats.

Integration rules for various topologies are described under `integrules`. Vertex coordinates of the reference element can be found using an `integrules` command containing the name of the element such as `r1=integrules('q4p');r1.xi`.

Backward compatibility note : if no element property entry is defined, or with a `p_solid` entry with the integration rule set to zero, the element defaults to the historical 3D mechanic elements described in section 5.16.2.

See also `fe_mat`, `m_elastic`, `fe_mk`, `feplot`

See section 4.1 .

integrules

Purpose Command function for FEM integration rule support.

Description This function groups integration rule manipulation utilities used by various elements. The following calls generate the reference `EltConst` data structure (see section 5.14.1).

`Gauss`

This command supports the definition of Gauss points and associated weights. It is called with `integrules('Gauss Topology',RuleNumber)`. Supported topologies are `1d` (line), `q2d` (2D quadrangle), `t2d` (2D triangle), `t3d` (3D tetrahedron), `p3d` (3D prism), `h3d` (3D hexahedron). `integrules('Gauss q2d')` will list available 2D quadrangle rules. `-3` is always the default rule for the order of the element, `-2` a rule at nodes and `-1` the rule at center.

```
[ -3]    [ 0x1  double]    'element dependent default'
[ -2]    [ 0x1  double]    'node'
[ -1]    [ 1x4 double]    'center'
[102]    [ 4x4 double]    'gefdyn 2x2'
[  2]    [ 4x4 double]    'standard 2x2'
[109]    [ 9x4 double]    'Q4WT'
[103]    [ 9x4 double]    'gefdyn 3x3'
[104]    [16x4 double]    'gefdyn 4x4'
[  9]    [ 9x4 double]    '9 point'
[  3]    [ 9x4 double]    'standard 3x3'
[  2]    [ 4x4 double]    'standard 2x2'
[ 13]    [13x4 double]    '2x2 and 3x3'
```

`bar1,beam1,beam3`

For integration rule selection, these elements use the 1D rules whose list you can find using `integrules('Gauss1d')`.

Geometric orientation convention for segment is • (1) → (2)

One can show the edge using `elt_name edge` (e.g. `beaml edge`).

`t3p,t6p`

Vertex coordinates of the reference element can be found using `r1=integrules('tria3');r1.xi`.

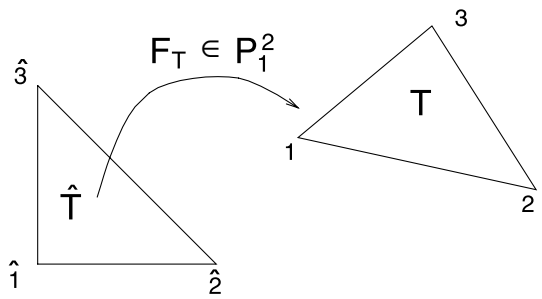


Figure 6.1: `t3p` reference element.

Vertex coordinates of the reference element can be found using `r1=integrules('tria6');r1.xi`.

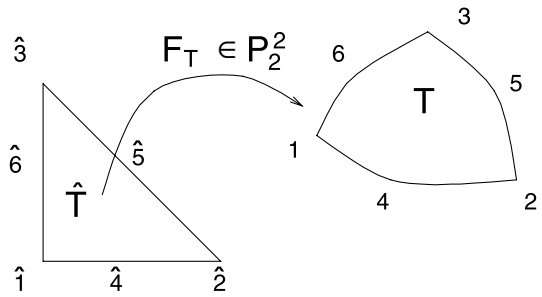


Figure 6.2: `t6p` reference element.

For integration rule selection, these elements use the 2D triangle rules whose list you can find using `integrules('Gausst2d')`.

Geometric orientation convention for triangle is to number anti-clockwise in the two-dimensional case (in the three-dimensional case, there is no orientation).
• edge [1]: (1) → (2) (nodes 4, 5,... if there are supplementary nodes) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (1) (...)

One can show the edges or faces using `elt_name edge` or `elt_name face` (e.g. `t3p edge`).

`q4p, q5p, q8p`

Vertex coordinates of the reference element can be found using `r1=integrules('quad4');r1.xi`.

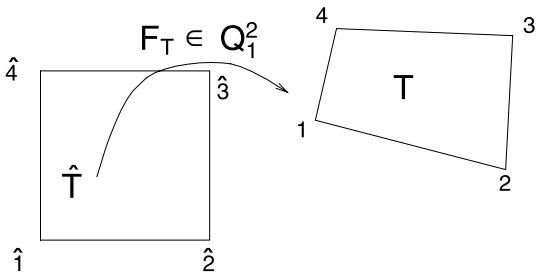


Figure 6.3: `q4p` reference element.

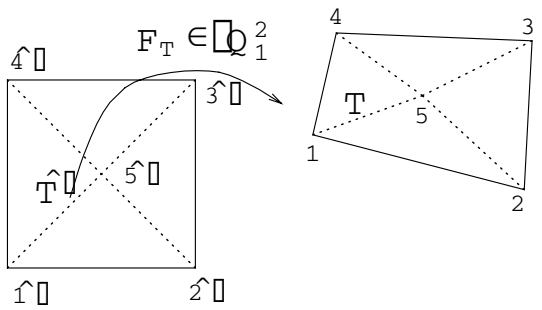


Figure 6.4: `q5p` reference element.

Vertex coordinates of the reference element can befound using the `r1=integrules('quadb');r1.xi`.

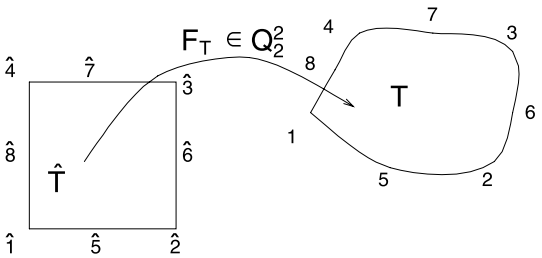


Figure 6.5: `q8p` reference element.

For integration rule selection, these elements use the 2D quadrangle rules whos list you can find using `integrules('Gaussq2d')`.

Geometric orientation convention for quadrilateral is to number anti-clockwise (same remark as for the triangle)
• edge [1]: (1) → (2) (nodes 5, 6, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (4) • edge [4]: (4) → (1)

One can show the edges or faces using `elt_name edge` or `elt_name face` (e.g. `q4p edge`).

`tetra4,tetra10`

3D tetrahedron geometries with linear and quadratic shape functions. Vertex coordinates of the reference element can be found using `r1=integrules('tetra4');r1.xi` (command `'tetra10'`).

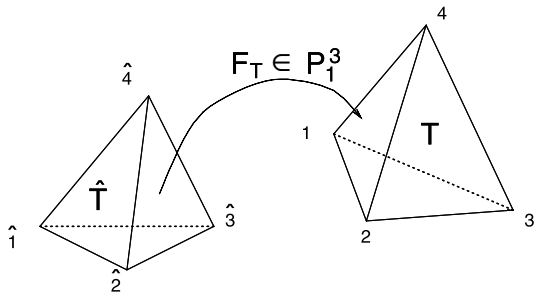


Figure 6.6: `tetra4` reference element.

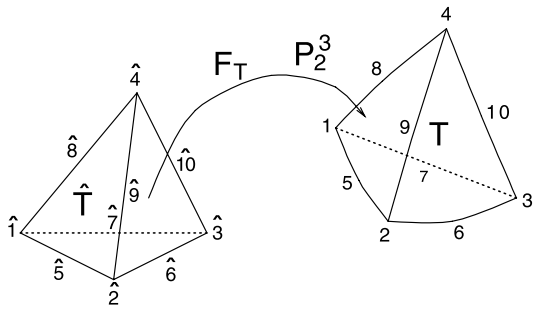


Figure 6.7: `tetra10` reference element.

For integration rule selection, these elements use the 3D pentahedron rules whose list you can find using `integrules('Gausst3d')`.

Geometric orientation convention for tetrahedron is to have trihedral $(\vec{12}, \vec{13}, \vec{14})$ direct ($\vec{ij}$ designates the vector from point i to point j).
• edge [1]: (1) → (2) (nodes 5, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (1)
• edge [4]: (1) → (4) • edge [5]: (2) → (4) • edge [6]: (3) → (4) (nodes ..., p)

All faces, seen from the exterior, are described anti-clockwise:
• face [1]: (1) (3) (2) (nodes p+1, ...) • face [2]: (1) (4) (3) (...)
• face [3]: (1) (2) (4) • face [4]: (2) (3) (4)

One can show the edges or faces using `elt_name edge` or `elt_name face` (e.g. `tetra10 face`).

`penta6, penta15`

3D prism geometries with linear and quadratic shape functions. Vertex coordinates of the reference element can be found using `r1=integrules('penta6');r1.xi` (or command `'penta15'`).

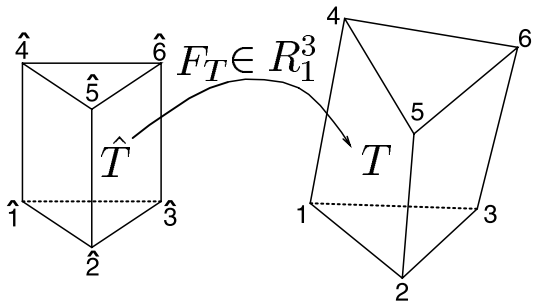


Figure 6.8: `penta6` reference element.

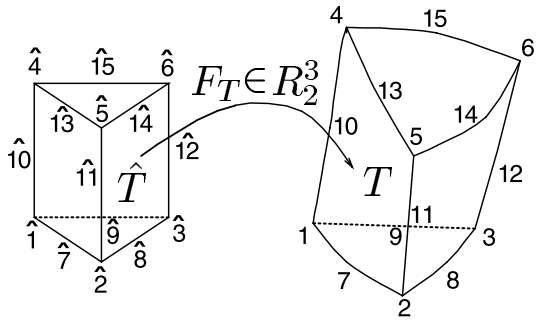


Figure 6.9: `penta15` reference element.

For integration rule selection, these elements use the 3D pentahedron rules whose list you can find using `integrules('Gaussp3d')`.

Geometric orientation convention for pentahedron is to have trihedral $(\vec{12}, \vec{13}, \vec{14})$ direct
• edge [1]: (1) → (2) (nodes 7, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (1)
• edge [4]: (1) → (4) • edge [5]: (2) → (5) • edge [6]: (3) → (6)
• edge [7]: (4) → (5) • edge [8]: (5) → (6) • edge [9]: (6) → (4) (nodes ..., p)

All faces, seen from the exterior, are described anti-clockwise.
• face [1] : (1) (3) (2) (nodes p+1, ...) • face [2] : (1) (4) (6) (3) • face [3] : (1) (2) (5) (4)
• face [4] : (4) (5) (6) • face [5] : (2) (3) (5) (6)

One can show the edges or faces using `elt_name edge` or `elt_name face` (e.g. `penta15 face`).

`hexa8, hexa20, hexa21, hexa27`

3D brick geometries, using linear `hexa8`, and quadratic shape functions. Vertex coordinates of the reference element can be found using `r1=integrules('hexa8');r1.xi` (or command `'hexa20'`, `'hexa27'`).

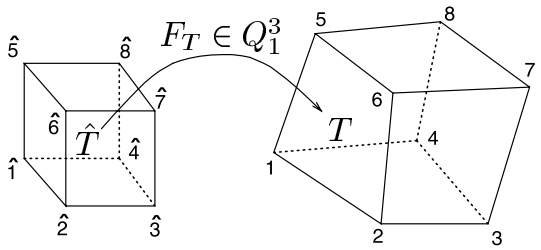


Figure 6.10: `hexa8` reference topology.

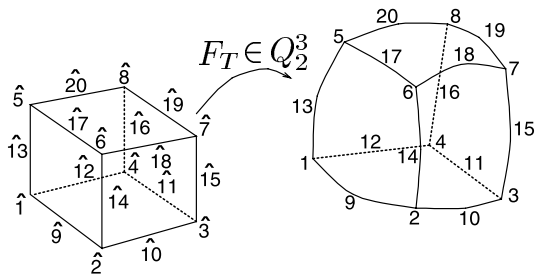


Figure 6.11: `hexa20` reference topology.

For integration rule selection, these elements use the 3D hexahedron rules whose list you can find using `integrules('Gausssh3d')`.

Geometric orientation convention for hexahedron is to have trihedral $(\vec{12}, \vec{14}, \vec{15})$ direct
• edge [1]: (1) → (2) (nodes 9, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (4)
• edge [4]: (4) → (1) • edge [5]: (1) → (5) • edge [6]: (2) → (6)
• edge [7]: (3) → (7) • edge [8]: (4) → (8) • edge [9]: (5) → (6)
• edge [10]: (6) → (7) • edge [11]: (7) → (8) • edge [12]: (8) → (5) (nodes ..., p)

All faces, seen from the exterior, are described anti-clockwise.
• face [1] : (1) (4) (3) (2) (nodes p+1, ...) • face [2] : (1) (5) (8) (4)
• face [3] : (1) (2) (6) (5) • face [4] : (5) (6) (7) (8)
• face [5] : (2) (3) (7) (6) • face [6] : (3) (4) (8) (7)

One can show the edges or faces using `elt_name edge` or `elt_name face` (e.g. `hexa8 face`).

BuildNDN

The commands are extremely low level utilities to fill the .NDN field for a given set of nodes. The calling format is of_mk('BuildNDN',type,rule,nodeE) where type is an int32 that specifies the rule to be used : 2 for 2D, 3 for 3D, 31 for 3D with xyz sorting of NDN columns, 23 for surface in a 3D model, 13 for a 3D line. A negative value can be used to switch to the .m file implementation in integrules.

The 23 rule generates a transformation with the first axis along N, r, the second axis orthogonal in the plane tangent to N, r, N, s and the third axis locally normal to the element surface. If a local material orientation is provided in columns 5 to 7 of nodeE then the material x axis is defined by projection on the surface.

With the 32 rule if a local material orientation is provided in columns 5 to 7 for x and 8 to 10 for y the spatial derivatives of the shape functions are given in this local frame.

The rule structure is described earlier in this section and node has three columns that give the positions in the of nodes of the current element. The rule.NDN and rule.jdet fields are modified. They must have the correct size before the call is made or severe crashes can be experienced.

If a rule.bas field is defined $(9 \times Nw)$, each column is filled to contain the local basis at the integration point for 23 and 13 types. If a rule.J field with $(4 \times Nw)$, each column is filled to contain the jacobian at the integration point for 23.

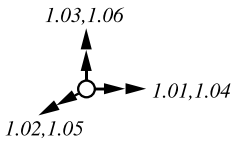
```
model=femesh('testhexa8');  nodeE=model.Node(:,5:7);
opt=integrules('hexa8',-1);
nodeE(:,5:10)=0; nodeE(:,7)=1;  nodeE(:,8)=1; % xe=z and ye=y
integrules('buildndn',32,opt,nodeE)

model=femesh('testquad4'); nodeE=model.Node(:,5:7);
opt=integrules('q4p',-1);opt.bas=zeros(9,opt.Nw);opt.J=zeros(4,opt.Nw);
nodeE(:,5:10)=0; nodeE(:,5:6)=1;  % xe= along [1,1,0]
integrules('buildndn',23,opt,nodeE)
```

See also elem0

mass1,mass2

Purpose Concentrated mass elements.

Description



`mass1` places a diagonal concentrated mass and inertia at one node.

In a model description matrix, **element property rows** for `mass1` elements follow the format

```
[NodeID mxx myy mzz ixx iyy izz EltID]
```

where the concentrated nodal mass associated to the DOFs `.01` to `.06` of the indicated node is given by

```
diag([mxx myy mzz ixx iyy izz])
```

Note `feutil GetDof` eliminates DOFs where the inertia is zero. You should thus use a small but non zero mass to force the use of all six DOFs.

For `mass2` elements, the **element property rows** follow the format

```
[n1 M I11 I21 I22 I31 I32 I33 EltID CID X1 X2 X3 MatId ProId]
```

which, for no offset, corresponds to matrices given by

$$\begin{bmatrix} M & & \text{symmetric} & & \\ & M & & & \\ & & M & & \\ & & & I_{11} & \\ & & & -I_{21} & I_{22} \\ & & & -I_{31} & -I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} \int \rho dV & & \text{symmetric} & & \\ & \int \rho dV & & & \\ & & \int \rho dV & & \\ & & & \int \rho(x^2+y^2)dV & \\ & & & -I_{21} & I_{22} \\ & & & -I_{31} & -I_{32} & I_{33} \end{bmatrix}$$

Note that local coordinates `CID` are not currently supported by `mass2` elements.

See also `femesh`, `feplot`

m_elastic

Purpose Material function for elastic solids and fluids.

Syntax

```
mat= m_elastic('default')
mat= m_elastic('database name')
pl = m_elastic('dbval MatId name');
pl = m_elastic('dbval -unit TM MatId name');
```

Description This help starts by describing the main commands of `m_elastic` : `Database` and `Dbval`. Materials formats supported by `m_elastic` are are then described.

`[Database,Dbval] [-unit TY] [,MatiD]] Name`

A material property function is expected to store a number of standard materials. See section 5.3 for material property interface.

`m_elastic('database Steel')` returns a the data structure describing steel. `m_elastic('dbval 100 Steel')` only returns the property row.

```
% List of materials in data base
m_elastic info
% examples of row building and conversion
pl=m_elastic([100 fe_mat('m_elastic','SI',1) 210e9 .3 7800], ...
  'dbval 101 aluminum', ...
  'dbval 200 lamina .27 3e9 .4 1200 0  790e9 .3 1780 0');
pl=fe_mat('convert SITM',pl);
pl=m_elastic(pl,'dbval -unit TM 102 steel')
```

You can generate orthotropic shell properties using the `Dbval 100 lamina VolFrac Ef nu_f rho_f G_m E_m nu_m Rho_m G_m` command which gives fiber and matrix characteristics as illustrated above.

The default material is steel.

Subtypes `m_elastic` supports the following material subtypes

`1 : standard isotropic`

Standard isotropic materials, see section 3.2.1 and section 3.2.2, are described by a row of the form

```
[MatID   typ   E nu rho G eta alpha T0]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',1)` command, E (Young's modulus), ν (Poisson's ratio), ρ (density), G (shear modulus, set to $G = E/2(1+\nu)$ if equal to zero). η loss factor for hysteretic damping modeling. α thermal expansion coefficient. T_0 reference temperature.

`2 : acoustic fluid`

Acoustic fluid , see section 3.2.3,are described by a row of the form

```
[MatId typ rho C eta]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',2)` command, ρ (density), C (velocity) and η (loss factor). The bulk modulus is then given by $K = \rho C^2$.

`3 : 3-D anisotropic solid`

3-D Anisotropic solid, see section 3.2.1, are described by a row of the form

```
[MatId typ Gij rho eta]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',3)` command, *rho* (density), *eta* (loss factor) and *Gij* a row containing

```
[G11 G12 G22 G13 G23 G33 G14 G24 G34 G44 ...
 G15 G25 G35 G45 G55 G16 G26 G36 G46 G56 G66]
```

`4 : 2-D anisotropic solid`

2-D Anisotropic solid, see section 3.2.2, are described by a row of the form

```
[MatId typ E11 E12 E22 E13 E23 E33 rho eta a1 a2 a3]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',4)` command, *rho* (density), *eta* (loss factor) and *Eij* elastic constants and *ai* anisotropic thermal expansion coefficients.

`5 : shell orthotropic material`

shell orthotropic material, see section 3.2.4, are described by a row of the form

```
[MatId typ E1 E2 nu12 G12 G13 G23 Rho A1 A2 TREF Xt Xc Yt Yc S Ge ...
  F12 STRN]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',5)` command, *rho* (density), ...

See also Section 3.6.4, section 5.3, `fe_mat`, `p_shell`

m_hyper

Purpose Material function for hyperelastic solids.

Syntax

```
mat= m_hyper('default')
mat= m_hyper('database name')
pl = m_hyper('dbval MatId name');
pl = m_hyper('dbval -unit TM MatId name');
```

Description Function based on `m_elastic` function adapted for hyperelastic material. Only subtype 1 is currently used:

`1 : Nominal hyperelastic material`

Nominal hyperelastic materials are described by a row of the form

`[MatID typ rho Wtype C_1 C_2 K]`

with `typ` an identifier generated with the `fe_mat('m_hyper','SI',1)` command, *rho* (density), *Wtype* (value for Energy choice), C_1, C_2, K (energy coefficients). Possible values for *Wtype* are:

$$0: \quad W = C_1(J_1-3) + C_2(J_2-3) + K(J_3-1)^2$$
$$1: \quad W = C_1(J_1-3) + C_2(J_2-3) + K(J_3-1) - (C_1+2C_2+K)\ln(J_3)$$

Other energy functions can be added by editing the `hyper.c Enpassiv` function.

In RivlinCube test, m_hyper is called in this form:

```
model.pl=m_hyper('dbval 100 Ref'); % this is where the material is defined
```

the hyperelastic material called "Ref" is described in the database of `m_hyper.m` file:

```
out.pl=[MatId fe_mat('type','m_hyper','SI',1) 1e-06 0 .3 .2 .3];
out.name='Ref';
out.type='m_hyper';
out.unit='SI';
```

Here is an example to set your material property for a given structure model:

```
model.pl = [MatID fe_mat('m_hyper','SI',1) typ rho Wtype C_1 C_2 K];
model.Elt(2:end,length(feval(ElemF,'node')+1)) = MatID;
```

p_beam

Purpose Element property function for beams

Syntax

```
il = p_beam('default')
il = p_beam('database','name')
il = p_beam('dbval ProId','name');
il = p_beam('dbval -unit TM ProId name');
```

Description This help starts by describing the main commands : `p_beam Database` and `Dbval`. Supported `p_beam` subtypes and their formats are then described.

[Database,Dbval] ...

`p_beam` contains a number of defaults obtained with `p_beam('database')` or `p_beam('dbval MatId')`. You can select a particular entry of the database with using a name matching the database entries. You can also automatically compute the properties of standard beams

`circle r`	beam with full circular section of radius `r`
`rectangle b h`	beam with full rectangular section of width `b` and height `h`.
`NastranCrossSection Dimi`	Some nastran cross sections are available (see list in subtype 3 section). For example `p_beam('database ROD 1.1')`.

`p_beam('database reftube')` gives a reference property of subtype 3 for a tube.

For example, you will obtain the section property row with `EltId` 100 associated with a circular cross section of $0.05m$ or a rectangular $0.05 \times 0.01m$ cross section using

```
pro = p_beam('database 100 rectangle .05 .01')
il = p_beam(pro.il,'dbval 101 circle .05')
il(end+1,1:6)=[102 fe_mat('p_beam','SI',1) 0 0 0 1e-5];
il = fe_mat('convert SITM',il);
il = p_beam(il,'dbval -unit TM 103 rectangle .05 .01')
```

Beam format description and subtypes

Element properties are described by the row of an element property matrix or a data structure with an `.il` field containing this row (see section 5.4). Element property functions such as `p_beam` support graphical editing of properties and a database of standard properties.

For a tutorial on material/element property handling see section 3.6.4. For a programmers reference on formats used to describe element properties see section 5.4.

1 : standard

```
[ProID   type   J I1 I2 A   k1 k2 Lump NSM]
```

ProID	element property identification number
type	identifier obtained with fe_mat('p_beam','SI',1)
J	torsional stiffness parameter (often different from polar moment of inertia I1+I2)
I1	moment of inertia for bending plane 1 defined by a third node nr or the vector vx vy vz. For a case with a beam along x and plane 1 the xy plane I1 is equal to $Iz = \int_S y^2 ds$.
I2	moment of inertia for bending plane 2 (containing the beam and orthogonal to plane 1.
A	section area
k1	(optional) shear factor for motion in plane 1 (when not 0, a Timoshenko beam element is used). The effective area of shear is given by $k_1 A$.
k2	(optional) shear factor for direction 2
lump	(optional) request for lumped mass model if set to 1
NSM	(optional) non structural mass (density per unit length)

bar1 elements only use the section area. All other parameters are ignored.

beam1 elements use all parameters. Without correction factors (*k1 k2* not given or set to 0), the beam1 element is the standard Bernoulli-Euler 12 DOF element based on linear interpolations for traction and torsion and cubic interpolations for flexion (see Ref. [3] for example). When non zero shear factors are given, the bending properties are based on a Timoshenko beam element with selective reduced integration of the shear stiffness [4]. No correction for rotational inertia of sections is used.

3 : Cross section database

This subtype can be used to refer to standard cross sections defined in database. It is particularly used by nasread when importing NASTRAN PBEAML properties.

```
[ProID   type   0  Section Dim1 ...]
```

ProID	element property identification number
type	identifier obtained with fe_mat('p_beam','SI',3)
Section	identifier of the cross section obtained with comstr('*SectionName*',-32) where *SectionName* is a string defining the section (see below).
Dim1 ...	dimensions of the cross section.

Cross section, if existing, is compatible with NASTRAN PBEAML definition. Equivalent moment of inertia and tensional stiffness are computed at the centroid of the section. Currently available sections are ROD (1 dim), TUBE (2 dims), T (4 dims), I (6 dims), BAR (2 dims), CHAN1 (4 dims), CHAN2 (4 dims).

See also Section 3.6.4, section 5.4, fe_mat

p_heat

Purpose Formulation and material support for the heat equation

Syntax

```
il = p_heat('default')
```

Description This help starts by describing the main commands : p_heat Database and Dbval. Supported p_heat subtypes and their formats are then described.

[Database,Dbval] ...

p_heat database

```
il=p_heat('database');
```

Heat equation element properties

Element properties are described by the row of an element property matrix or a data structure with an .il field containing this row (see section 5.4). Element property functions such as p_beam support graphical editing of properties and a database of standard properties.

For a tutorial on material/element property handling see section 3.6.4. For a programmers reference on formats used to describe element properties see section 5.4.

1 : Volume element for heat diffusion (dimension DIM)
[ProId fe_mat('p_heat','SI',1) CordM Integ DIM]

ProID	element property identification number
type	identifier obtained with fe_mat('p_beam','SI',1)
Integ	is rule number in integrules
DIM	is problem dimension 2 or 3 D

2 : Surface element for heat exchange (dimension DIM-1)
[ProId fe_mat('p_heat','SI',2) CordM Integ DIM]

ProID	element property identification number
type	identifier obtained with fe_mat('p_beam','SI',1)
Integ	is rule number in integrules
DIM	is problem dimension 2 or 3 D

1 : Heat equation material
[MatId fe_mat('m_heat','SI',2) k rho C alpha]

2D validation

Consider a bi-dimentinal annular thick domain Ω with radii $r_e = 1$ and $r_i = 0.5$. The data are specified on the internal circle Γ_i ans on the external circle Γ_e. The solid is made of homogeneous isotropic material, and its conductivity tensor thus reduces to a constant k. The steady state temperature distribution is then given by

$$-k\Delta\theta(x,y) = f(x,y) \quad in \quad \Omega. \tag{6.7}$$

The solid is subject to the following boundary conditions

- $\Gamma_i\,(r = r_i)$: Neumann condition

$$\frac{\partial\theta}{\partial n}(x,y) = g(x,y) \tag{6.8}$$

- $\Gamma_e\,(r = r_e)$: Dirichlet condition

$$\theta(x,y) = \theta_{ext}(x,y) \tag{6.9}$$

In above expressions, f is an internal heat source, θ_{ext} an external temperature at $r = r_e$, and g a fonction. All the variables depend on the variable x and y.

The OpenFEM model for this example can be found in `ofdemos('AnnularHeat')`. **Numerical application** : assuming $k = 1$, $f = 0$, $\alpha = 1e^{-10}$, $\theta_{ext}(x,y) = \exp(x)\cos(y)$ and $g(x,y) = -\dfrac{\exp(x)}{r_i}\left(\cos(y)x - \sin(y)x\right)$, the solution of the problem is given by

$$\theta(x,y) = \exp(x)\cos(y)$$

See also Section 3.6.4, section 5.4, `fe_mat`

p_shell

Purpose Element property function for shells

Syntax

```
il = p_shell('default');
il = p_shell('database ProId name');
il = p_shell('dbval ProId name');
il = p_shell('dbval -unit TM ProId name');
il = p_shell('SetDrill 0',il);
```

Description This help starts by describing the main commands : `p_shell Database` and `Dbval`. Supported `p_shell` subtypes and their formats are then described.

[Database,Dbval] ...

`p_shell` contains a number of defaults obtained with the `database` and `dbval` commands which respectively return a structure or a element property row. You can select a particular entry of the database with using a name matching the database entries.

You can also automatically compute the properties of standard shells with

`kirchhoff e`	Kirchhoff shell of thickness `e`
`mindlin e`	Mindlin shell of thickness `e`
`laminate MatIdi Ti Thetai`	Specification of a laminate property by giving the different ply `MatId`, thickness and angle.

You can append a string of the form `-f i` to select the appropriate shell formulation. For example, you will obtain the element property row with `EltId` 100 associated with a .1 thick Kirchhoff shell (with formulation 5) or the corresponding Mindlin plate use

```
il = p_shell('database 100 MindLin .1')
il = p_shell('dbval 100 kirchhoff .1 -f5')
il = p_shell('dbval 100 laminate 110 3e-3 30 110 3e-3 -30')
il = fe_mat('convert SITM',il);
il = p_shell(il,'dbval -unit TM 2 MindLin .1')
```

For laminates, you specify for each ply the `MatId`, thickness and angle.

Shell format description and subtypes

Element properties are described by the row of an element property matrix or a data structure with an `.il` field containing this row (see section 5.4). Element property functions such as `p_shell` support graphical editing of properties and a database of standard properties.

For a tutorial on material/element property handling see section 3.6.4. For a programmers reference on formats used to describe element properties see section 5.4.

`p_shell` currently only supports two subtypes

1 : standard isotropic

[ProID type f d 0 h k MID2 12I/T3 MID3 NSM Z1 Z2 MID4]

type	identifier obtained with `fe_mat('p_shell','SI',1)`
f 0	default, for other formulations the specific help for each element (quad4, ...)
d -1	no drilling stiffness. The element DOFs are the standard translations and rotations at all nodes (DOFs .01 to .06). The drill DOF (rotation .06 for a plate in the xy plane) has no stiffness and is thus eliminated by fe_mk if it corresponds to a global DOF direction. The default is d=1 (d is set to 1 for a declared value of zero).
d	arbitrary drilling stiffness with value proportional to d is added. This stiffness is often needed in shell problems but may lead to numerical conditioning problems if the stiffness value is very different from other physical stiffness values. Start with a value of 1. Use `il=p_shell('SetDrill d',il)` to set to d the drilling stiffness of all p_shell subtype 1 rows of the property matrix il.
h	plate thickness
k *k*	shear correction factor (default 5/6, default used if k is zero). This correction is not used for formulations based on triangles since tria3 is a thin plate element.
12I/T3	Ratio of bending moment of inertia to nominal T3/12 (default 1).
NSM	Non structural mass per unit area.
MID2	unused
MID3	unused
z1,z2	(unused) offset for fiber computations
MID4	unused

Shell strain is defined by the membrane, curvature and transverse shear (display with `p_shell('ConstShell')`).

$$\begin{Bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy} \\ \kappa_{xx} \\ \kappa_{yy} \\ 2\kappa_{xy} \\ \gamma_{xz} \\ \gamma_{yz} \end{Bmatrix} = \begin{bmatrix} N,x & 0 & 0 & 0 & 0 \\ 0 & N,y & 0 & 0 & 0 \\ N,y & N,x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -N,x \\ 0 & 0 & 0 & N,y & 0 \\ 0 & 0 & 0 & N,x & -N,y \\ 0 & 0 & N,x & 0 & N \\ 0 & 0 & N,y & -N & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \\ ru \\ rw \end{Bmatrix} \tag{6.10}$$

2 : composite

[ProID type Z0 NSM SB FT TREF GE LAM MatId1 T1 Theta1 SOUT1 ...]

p_shell

ProID	section property identification number
type	identifier obtained with fe_mat('p_shell','SI',2)
Z0	distance from reference plate to bottom surface.
NSM	non structural mass per unit area
SB	allowable shear stress of the bonding material
FT	Failure theory
TREF	Reference temperature
GE	Hysteretic loss factor
LAM	Laminate type
MatId*i*	MatId for ply *i*
T*i*	Thickness of ply *i*
Theta*i*	Orientation of ply *i*
SOUT*i*	Stress output request for ply *i*

Note that this subtype is based on the format used by NASTRAN for PCOMP but not currently implemented in any element. You can use the DbvalLaminate commands to generate standard entries.

$$\left\{ \begin{array}{c} N \\ M \\ Q \end{array} \right\} = \left[\begin{array}{ccc} A & B & 0 \\ B & D & 0 \\ 0 & 0 & F \end{array} \right] \left\{ \begin{array}{c} \epsilon \\ \kappa \\ \gamma \end{array} \right\} \tag{6.11}$$

See also Section 3.6.4, section 5.4, fe_mat

p_solid

Purpose Element property function for volume elements.

Syntax

```
il=p_solid('default')
il=p_solid('database ProId Value')
il=p_solid('dbval ProId Value')
il=p_solid('dbval -unit TM ProId name');
```

Description This help starts by describing the main commands : `p_solid Database` and `Dbval`. Supported `p_solid` subtypes and their formats are then described.

`[Database,Dbval] ...`

Element properties are described by the row of an element property matrix or a data structure with an `.il` field containing this row (see section 5.4). Element property functions such as `p_solid` support graphical editing of properties and a database of standard properties.

Accepted value in commands for the database are

- `d3 2` : 2x2x2 integration rule for linear volumes (hexa8 ...)
- `d3 -3` : default integration for all 3D elements
- `d3 3` : 3x3x3 integration rule for quadratic volumes (hexa20 ...)
- `d2 2` : 2x2x2 integration rule for linear volumes (q4p ...). You can also use `d2 1 0 2` for plane stress, and `d2 2 0 2` for axisymetry.
- `d2 3` : 3x3x3 integration rule for quadratic volumes (q8p ...)

For fixed values, use `p_solid('info')`.

For a tutorial on material/element property handling see section 3.6.4. For a programmers reference on formats used to describe element properties see section 5.4.

Examples of database property construction

```
il=p_solid([100 fe_mat('p_solid','SI',1) 0 3 0 2], ...
           'dbval 101 Full 2x2x2','dbval 102 d3 -3');
il=fe_mat('convert SITM',il);
il=p_solid(il,'dbval -unit TM 2 Reduced shear')
```

`Subtype 1 :  3D volume element`

```
[ProID fe_mat('p_solid','SI',1) Coordm In Stress Isop ]
```

p_solid

ProID	Property identification number
Coordm	Identification number of the material coordinates system (not used yet)
In	Integration rule selection (see integrules Gauss). 0 selects the legacy 3D mechanics element (of_mk_pre.c), -3 the default rule.
Stress	Location selection for stress output (NOT USED)
Isop	Integration scheme (will be used to select shear protection mechanims)

The underlying physics for this subtype are selected through the material property. Examples are 3D mechanics with m_elastic, , heat equation (p_heat).

```
Subtype 2 :  2D volume element
    [ProId fe_mat('p_solid','SI',2)  Form N In]
```

ProID	Property identification number
Type	Identifier obtained with fe_mat('p_solid,'SI',2)
Form	Formulation (0 plane strain, 1 plane stress, 2 axisymetric), see details in m_elastic.
N	Fourier harmonic for axisymetric elements that support it
In	Integration rule selection (see integrules Gauss). 0 selects legacy 2D element, -3 the default rule.

The underlying physics for this subtype are selected through the material property. Examples are 2D mechanics with m_elastic.

```
Subtype 3 :  ND-1 coupling element
    [ProId fe_mat('p_solid','SI',2) Integ Form Ndof1 ...]
```

ProID	Property identification number
Type	Identifier obtained with fe_mat('p_solid,'SI',3)
Integ	Integration rule selection (see integrules Gauss). 0 or -3 selects the default for the element.
Form	1 volume force, 2 volume force proportionnal to density, 3 pressure, 4: fluid/structure coupling, see fsc, 5 2D volume force, 6 2D pressure

See also Section 3.6.4, section 5.4, fe_mat

p_spring

Purpose Element property function for spring and rigid elements

Syntax

```
il=p_spring('default')
il=p_spring('database MatId Value')
il=p_spring('dbval MatId Value')
il=p_spring('dbval -unit TM ProId name');
```

Description This help starts by describing the main commands : p_spring Database and Dbval. Supported p_spring subtypes and their formats are then described.

[Database,Dbval] ...

Element properties are described by the row of an element property matrix or a data structure with an .il field containing this row (see section 5.4).

Examples of database property construction

```
il=p_spring('database 100 1e12 1e4 0')
il=p_spring('dbval 100 1e12');
il=fe_mat('convert SITM',il);
il=p_spring(il,'dbval 2 -unit TM 1e12')
```

p_spring currently supports 2 subtypes

1 : standard

[ProID type k m c Eta S]

ProID	property identification number
type	identifier obtained with fe_mat('p_spring','SI',1)
k	stiffness value
m	mass value
c	viscous damping value
eta	loss factor
S	Stress coefficient

2 : bush

[ProId Type k11:k66 c11:c66 Eta SA ST EA ET m v]

p_spring

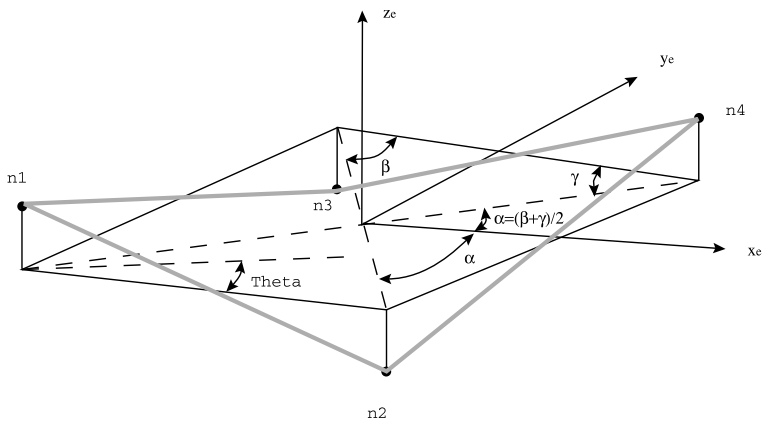
ProID	property identification number
type	identifier obtained with fe_mat('p_spring','SI',2)
ki	stiffness for each direction
ci	viscous damping for each direction
SA	stress recovery coef for translations
ST	stress recovery coef for rotations
EA	strain recovery coef for translations
ET	strain recovery coef for rotations
m	mass
v	volume

See also Section 3.6.4, section 5.4, fe_mat

quad4, quadb, mitc4

Purpose 4 and 8 node quadrilateral plate/shell elements.

Description



In a model description matrix, **element property rows** for quad4, quadb and mitc4 elements follow the standard format

```
[n1 ... ni MatID ProID EltID Theta Zoff T1 ... Ti]
```

giving the node identification numbers ni (1 to 4 or 8), material MatID, property ProID. Other **optional** information is EltID the element identifier, Theta the angle between material x axis and element x axis, Zoff the off-set along the element z axis from the surface of the nodes to the reference plane (use femesh orient command to check z-axis orientation), Ti the thickness at nodes (used instead of il entry, currently the mean of the Ti is used).

If n3 and n4 are equal, the tria3 element is automatically used in place of the quad4.

Isotropic materials are currently the only supported (this may change soon). Their declaration follows the format described in m_elastic. Element property declarations follow the format described p_shell.

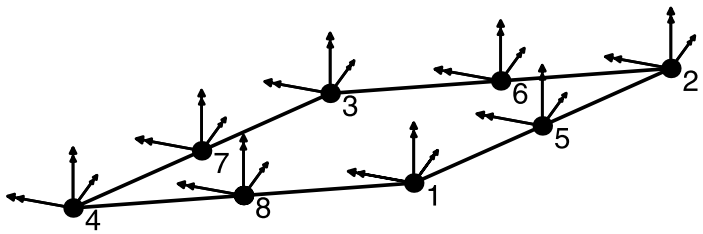
quad4

Supported formulations (il(3) see p_shell) are

- 1 4 tria3 thin plate elements with condensation of central node.
- 2 Q4WT for membrane and Q4gamma for bending. This is only applicable if the four nodes are in a single plane. When not, formulation 1 is called.
- 4 MITC4 calls the MITC4 element below. This implementation has not been tested extensively, so that the element may not be used in all configurations. It uses 5 DOFs per node with the two rotations being around orthogonal in-plane directions. This is not consistent for mixed element types assembly. Non

smooth surfaces are not handled properly because this is not implemented in the `feutil getnormals` command which is called for each group of `mitc4` elements.

`quadb`



Supported formulations (`il(3)` see `p_shell`) are

- `1` 8 tria3 thin plate elements with condensation of central node
- `2` isoparametric thick plate with reduced integration. For non-flat elements, formulation `1` is used.

See also `m_elastic`, `p_shell`, `fe_mk`, `feplot`

q4p, q8p, t3p, t6p and other 2D volumes

Purpose 2-D volume elements.

Description The q4p q5p, q8p, q9a, t3p, t6p elements are topology references for 2D volumes and 3D surfaces.

In a model description matrix, **element property rows** for this elements follow the standard format

```
[n1 ... ni MatID ProID EltID Theta]
```

giving the node identification numbers n1,...ni, material MatID, property ProID. Other **optional** information is EltID the element identifier, Theta the angle between material x axis and element x axis (material orientation maps are generally preferable).

These elements only define topologies, the nature of the problem to be solved should be specified using a property entry, see section 3.2 for supported problems and p_solid, p_heat, ... for formats.

Integration rules for various topologies are described under integrules. Vertex coordinates of the reference element can be found using an integrules command containing the name of the element such as r1=integrules('q4p');r1.xi.

Backward compatibility note : if no element property entry is defined, or with a p_solid entry with the integration rule set to zero, the element defaults to the historical 3D mechanic elements described in section 5.16.2.

These volume elements are used for various problem families.

See also fe_mat, fe_mk, feplot

rigid

Purpose Linearized rigid link constraints.

Synopsis

```
[T,cdof] = rigid(node,elt,mdof)
[T,cdof] = rigid(Up)
```

Description Rigid links are often used to model stiff connections in finite element models. One generates a set of linear constraints that relate the 6 DOFs of master M and slave S nodes by

$$\begin{Bmatrix} u \\ v \\ w \\ r_x \\ r_y \\ r_z \end{Bmatrix}_S = \begin{bmatrix} 1 & 0 & 0 & 0 & z_{MS} & -y_{MS} \\ 0 & 1 & 0 & -z_{MS} & 0 & x_{MS} \\ 0 & 0 & 1 & y_{MS} & -x_{MS} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \\ r_x \\ r_y \\ r_z \end{Bmatrix}_M$$

Although they are linear constraints rather than true elements, such connections can be declared using an element group of rigid connection with a header row of the form `[Inf abs('rigid')]` followed by as many element rows as connections of the form

```
 [ n1 n2 DofSel MatId ProId EltId]
```

where node `n2` will be rigidly connected to node `n1` which will remain free. `DofSel` lets you specify which of the 3 translations and 3 rotations are connected (thus `123` connects only translations while `123456` connects both translations and rotations).

The other strategy is to store them as a `case` entry. `rigid` entries are rows of the `Case.Stack` cell array giving `{'rigid', Name, Elt}`. `Name` is a string identifying the entry. `Elt` is a model description matrix containing `rigid` elements. The elements can also be declared as standard elements as in the following example which generates the mesh of a square plate with a rigid edge

```
femesh('reset');
femesh(';testquad4 divide 10 10;addsel');

% Define a rigid edge
femesh('selelt seledge & innode{x==0}')
femesh('setgroupa1 name rigid')
FEel0(femesh('findel0 group1'),3)=123456;
FEel0(femesh('findel0 group1'),4)=0;
model=fe_case(femesh,'rigid','Rigid edge',FEel0);

% Compute and display modes
def=fe_eig(model,[6 20 1e3]);
feplot(model,def); fecom(';view3;ch8;scd.1');
```

An additional call is `rigidAppend` in order to simply add new rigid links. You may use a list of the form `[MasterNode slaveDOF slaveNode_1 slaveNode_2 ... slaveNode_i]` or of the form of an element matrix (containing a header).

The preceding call would be

```
model=fe_case(femesh,'rigidAppend','Rigid edge',FEl0);
```

or

```
model=fe_case(femesh,'rigidAppend','Rigid edge',...
              [FEl0(2,2) .01 FEl0(2,2) FEl0(3,2)];
```

The `rigid` function is only used for low level access. High level use of constraints is discussed in section 5.13 which discusses handling of linear constraints in general.

If coordinate systems are defined in field `model.bas` (see `basis`), `PID` (position coordinate system) and `DID` (displacement coordinate system) declarations in columns 2 and 3 of `model.Node` are properly handled.

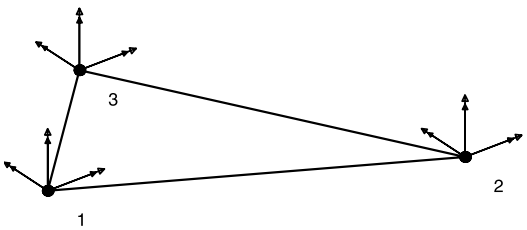
You can use penalized rigid links (`celas` element) instead of truly rigid connections. This requires the selection of a stiffness constant but can be easier to manipulate. To change a group of `rigid` elements into `celas` elements change the element group name `femesh('SetGroup rigid name celas')` and set the stiffness constant `FEelt(femesh('FindEltGroup`*i*`'),7) = Kv`.

See also Section 5.13, `celas`

tria3, tria6

Purpose Element functions for a 3 node/18 DOF and 6 nodes/36 DOF shell elements.

Description



In a model description matrix, **element property rows** for `tria3` elements follow the standard format

```
[n1 n2 n3 MatID ProID EltID Theta Zoff T1 T2 T3]
```

giving the node identification numbers `ni`, material `MatID`, property `ProID`. Other **optional** information is `EltID` the element identifier, `Theta` the angle between material x axis and element x axis (currently unused), `Zoff` the off-set along the element z axis from the surface of the nodes to the reference plane, `Ti` the thickness at nodes (used instead of `il` entry, currently the mean of the `Ti` is used).

The element only supports isotropic materials with the format described in `m_elastic`.

The supported property declaration format is described in `p_shell`. Note that `tria3` only supports thin plate formulations.

`tria3` uses a T3 triangle for membrane properties and a DKT for flexion (see [5] for example).

`tria6` is currently supported for plotting only.

See also `quad4`, `quadb`, `fe_mat`, `p_shell`, `m_elastic`, `fe_mk`, `feplot`

7

Function reference

This section contains detailed descriptions of the functions in *Structural Dynamics Toolbox*. It begins with a list of functions grouped by subject area and continues with the reference entries in alphabetical order. From MATLAB short text information is available through the `help` command while the HTML version of this manual can be accessed through `doc`.

For easier use, most functions have several optional arguments. In a reference entry under syntax, the function is first listed with all the necessary input arguments and then with all *possible* input arguments. Most functions can be used with any number of arguments between these extremes, the rule being that missing, trailing arguments are given default values, as defined in the manual.

As always in MATLAB, all output arguments of functions do not have to be specified, and are then not returned to the user.

As indicated in their synopsis some functions allow different types of output arguments. The different output formats are then distinguished by the number of output arguments, so that all outputs must be asked by the user.

Typesetting conventions and mathematical notations used in this manual are described in section 1.2.

Element functions are detailed in chapter s*eltfun.

A list of demonstrations is given in section **??**.

General Tools	
femesh	UI command function for mesh building and modification
feutil	mesh handling utilities
fe_load	creation of distributed load vectors
fe_stres	element energies and stress computations

FEM Utilities	
basis	coordinate transformation utilities
fe_c	DOF selection and I/O matrix creation
fe_mat	material property handling utilities
getegroup	Get element group header positions
m_elastic	elastic material handling
of_mk	Gateway function to the FORTRAN library
rigid	projection matrix for linearized rigid body constraints
p_beam	beam section property handling
p_shell	shell property handling
p_spring	spring property handling
readnopo	Read MODULEF nopo format
ofact	factored matrix object

Non Current SDT Functions	
comgui	GUI tools (commode, iicom also)
comstr	general purpose string handling routine
fecom	UI command function for deformations created with feplot
feplot	GUI for 3-D deformation plots
fe_eig	partial and full eigenvalue computations
fe_mk	assembly of full and reduced FE models
fe_super	general element utilities
iimouse	mouse related callbacks (zooming, info, ...)
iigui,fegui	global variable inits
phaseb	unwrapped phase
remi	integer remainder
sdtdef	defaults handling

Other Utilities	
ofutil	admnistration tools for OpenFEM
sdtw	extended warning handling
stack_get	.Stack field handling tools

basis

Purpose Coordinate system handling utilities

Syntax

```
p          = basis(x,y)
[node,bas]= basis(node,bas)
[bas,x]    = basis(node)
[ ... ]    = basis('Command', ... )
```

Description

`p = basis(x,y)`

Basis from nodes (typically used in element functions to determine local coordinate systems). `x` and `y` are two vectors of dimension 3 (for finite element purposes) which can be given either as rows or columns (they are automatically transformed to columns). The orthonormal matrix `p` is computed as follows

$$p = \left[\frac{\vec{x}}{\|\vec{x}\|}, \frac{\vec{y}_1}{\|\vec{y}_1\|}, \frac{\vec{x}\times\vec{y}_1}{\|\vec{x}\|\|\vec{y}_1\|}\right]$$

where $\vec{y}_1$ is the component of $\vec{y}$ that is orthogonal to $\vec{x}$

$$\vec{y}_1 = \vec{y} - \vec{x}\frac{\vec{x}^T\vec{y}}{\|\vec{x}\|^2}$$

If `x` and `y` are collinear `y` is selected along the smallest component of `x`. A warning message is passed unless a third argument exists (call of the form `basis(x,y,1)`).

`p = basis([2 0 0],[1 1 1])` gives the orthonormal basis matrix `p`

```
 p =
     1.0000         0         0
          0    0.7071    0.7071
          0    0.7071   -0.7071
```

`[nodeGlob,bas]=basis('nodebas',model)`

Local to global node transformation with recursive transformation of coordinate system definitions stored in `bas`. Column 2 in `nodeLocal` is assumed give displacement coordinate system identifiers `PID` matching those in the first column of `bas`. `[nodeGlobal,bas]= basis(nodeLocal,bas)` is an older acceptable format.

Coordinate systems are stored in a matrix where each row represents a coordinate system using any of the three formats

```
 CorID Type RefID A1   A2   A3   B1 B2 B3 C1 C2 C3 0  0  0  s
 CorID Type 0     NIdA NIdB NIdC 0  0  0  0  0  0  0  0  0  s
 CorID Type 0     Ax Ay Az       Ux Uy Uz Vx Vy Vz Wx Wy Wz s
```

Supported coordinate types are `1` rectangular, `2` cylindrical, `3` spherical. For these types, the nodal coordinates in the initial `nodeLocal` matrix are `x y z`, `r teta z`, `r teta phi` respectively.

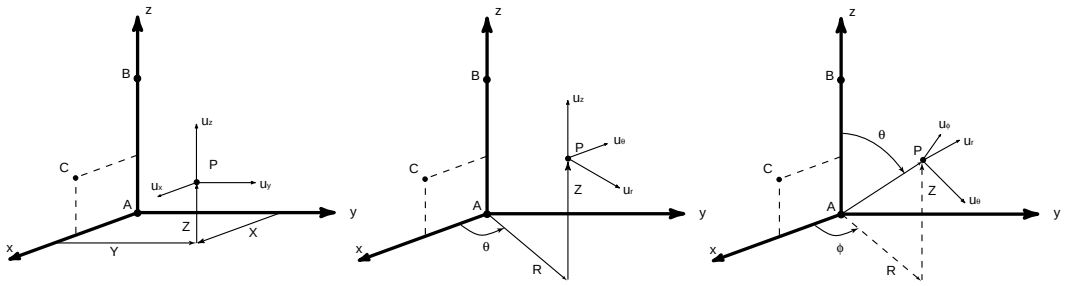


Figure 7.1: Coordinates convention.

The first format defines the coordinate system by giving the coordinates of three nodes `A`, `B`, `C` as shown in the figure above. These coordinates are given in coordinate system `RefID` which can be 0 (global coordinate system) or an another `CordId` in the list (recursive definition).

The second format specifies the same nodes using identifiers `NIdA`, `NIdB`, `NIdC` of nodes defined in `node`.

The last format gives, in the global reference system, the position `Ax Ay Az` of the origin of the coordinate system and the directions of the `x`, `y` and `z` axes.

The `s` scale factor can be used to define position of nodes using two different unit systems. This is used for test/analysis correlation. The scale factor has no effect on the definition of displacement coordinate systems.

```
cGL= basis('trans [ ,t][ ,l][,e]',bas,node,DOF)
```

The *transformation basis for displacement coordinate systems* is returned with this call. Column 3 in `node` is assumed give displacement coordinate system identifiers `DID` matching those in the first column of `bas`.

By default, `node` is assumed to be given in global coordinates. The `l` modifier is used to tell basis that the nodes are given in local coordinates.

Without the `DOF` input argument, the function returns a transformation defined at the 3 translation and 3 rotations at each node. The `t` modifier restricts the result to translations. With the `DOF` argument, the output is defined at DOFs in `DOF`. The `e` modifier returns a square transformation matrix.

```
nodeGlobal = basis('gnode',bas,nodeLocal)
```

Given a single coordinate system definition `bas`, associated nodes `nodeLocal` (with coordinates `x y z`, `r teta z`, `r teta phi` for Cartesian, cylindrical and spherical coordinate systems respectively) are transformed to the global Cartesian coordinate system. This is a low level command used for the global transformation `[node,bas] = basis(node,bas)`.

basis

```
[p,nodeL] = basis(node)
```

Element basis computation With two output arguments and an input `node` matrix, `basis` computes an appropriate local basis `bas` and node positions in local coordinates `x`. This is used by some element functions (`quad4`) to determine the element basis.

See also `beam1`, section 5.1,section 5.2
Note : the name of this function is in conflict with `basis` of the *Financial Toolbox*.

fecom

Purpose UI command function for the visualization of 3-D deformation plots.

Syntax

```
fecom
fecom CommandString
fecom('CommandString',AdditionalArgument)
```

Description The non current *SDT 3* version of this function is included in OpenFEM. Use the `help fecom` command to get help.

See also `feplot`, `femesh`

femesh, feutil

Purpose Finite element mesh handling utilities.

Syntax

```
femesh CommandString
femesh('CommandString')
[out,out1] = femesh('CommandString',in1,in2)
```

Description `femesh` and `feutil` provide a number of tools for mesh creation and manipulation. `feutil` requires all arguments to be provided while `femesh` uses global variables to define the proper object of which to apply a command. `femesh` uses the following *standard global variables* which are declared as global in your workspace when you call `femesh`

`FEnode`	main set of nodes
`FEn0`	selected set of nodes
`FEn1`	alternate set of nodes
`FElt`	main finite element model description matrix
`FEl0`	selected finite element model description matrix
`FEl1`	alternate finite element model description matrix

By default, `femesh` automatically uses base workspace definitions of the standard global variables (even if they are not declared as global). When using the standard global variables within functions, you should always declare them as global at the beginning of your function. If you don't declare them as global modifications that you perform will not be taken into account, unless you call `femesh` from your function which will declare the variables as global there too. The only thing that you should avoid is to use `clear` (instead of `clear global`) within a function and then reinitialize the variable to something non-zero. In such cases the global variable is used and a warning is passed.

Available `femesh` commands are

`;`

Command chaining. Commands with no input (other than the command) or output argument, can be chained using a call of the form `femesh(';Com1;Com2')`. `commode` is then used for command parsing.

`Add FEel`*`i`* `FEel`*`j`*`, AddSel`

Combine two FE model description matrices. The characters *i* and *j* can specify any of the main `t`, selected `0` and alternate `1` finite element model description matrices. The elements in the model matrix `FEel`*`j`* are appended to those of `FEel`*`i`*.

`AddSel` is equivalent to `AddFEeltFEel0` which adds the selection `FEel0` to the main model `FElt`.

This is an example of the creation of `FElt` using 2 selections (`FEel0` and `FEel1`)

```
femesh('reset');
femesh('testquad4');                    % one quad4 created
femesh('divide',[0 .1 .2 1],[0 .3 1]);  % divisions
FEel0=FEel0(1:end-1,:);                 % suppress 1 element in FEel0
femesh('addsel');                       % add FEel0 into FEelt
FEel1=[Inf abs('tria3');9 10 12  1 1 0];% create FEel1
femesh('add FEelt FEel1');              % add FEel1 into FEelt
femesh plotelt                          % plot FEelt
```

AddNode [,New] [, From i] [,epsl *val*]

Combine, append (without/with new) FEn0 to FEnode. Additional uses of AddNode are provided using the format

```
[AllNode,ind]=femesh('AddNode',OldNode,NewNode);
```

which combines NewNode *to* OldNode. AddNode finds nodes in NewNode that coincide with nodes in OldNode and appends other nodes to form AllNode. ind gives the indices of the NewNode nodes in the AllNode matrix.

This function is also accessible using feutil. For example

```
[AllNode,ind]=feutil('AddNode',OldNode,NewNode);
```

NewNode can be specified as a matrix with three columns giving xyz coordinates. The minimal distance below which two nodes are considered identical is given by sdtdef epsl (default 1e-6).

`[AllNode,ind]=feutil('AddNode From 10000',OldNode,NewNode);` gives node numbers starting at 10000 for nodes in NewNode that are not in OldNode.

AddSet [NodeId, EltId, FaceId]

`model=feutil('AddSetNodeId',model,'name','FindNodeString')` adds the selection SelNodeString as a set of nodes name to model. Syntax is the same for AddSetEltId with a FindEltString selection.
The string `-id` *value* can be added to the command to specify a set ID.

Following example defines a set of each type on the ubeam model:

```
model=demosdt('demo ubeam');
cf=feplot
model=feutil('AddSetNodeId',model,'nodeset','z==1');
model=feutil('AddSetEltId -id18',model,'eltset','WithNode{z==0}');
[elt,ind]=feutil('findelt setname  eltset',model); % findelt based on set name
r1=cf.Stack{'eltset'};r1.type='FaceId';r1.data(:,2)=1;
cf.Stack{'set','faceset'}=r1;
r1=cf.Stack{'nodeset'};r1.type='DOF';r1.data=r1.data+0.02;
cf.Stack{'set','dofset'}=r1;
fecom(cf,'curtab Stack','eltset');
```

AddTest [,-EGID *i*][,*NodeShift*,Merge,Combine]

Combine test and analysis models. When combining test and analysis models you typically want to overlay a detailed finite element mesh with a coarse wire-frame representation of the test configuration. These models coming from different origins you will want combine the two models in FEelt.

By default the node sets are considered to be disjoint. New nodes are added starting from max(FEnode(:,1))+1 or from *NodeShift+1* if the argument is specified. Thus femesh('addtest *NodeShift*',TNode,TElt) adds test nodes TNode to FEnode while adding NodeShift to their initial identification number. The same NodeShift is added to node numbers in TElt which is appended to FEelt. TElt can be a wire frame matrix read with ufread.

With merge it is assumed that some nodes are common but their numbering is not coherent. femesh('addtest merge',NewNode,NewElt) can also be used to merge to FEM models. Non coincident nodes (as defined by the AddNode command) are added to FEnode and NewElt is renumbered according to the new FEnode.

With combine it is assumed that some nodes are common and their numbering is coherent. Nodes with new NodeId values are added to FEnode while common NodeId values are assumed to be located at the same positions.

You can specify an EGID value for the elements that are added using AddTest -EGID -1. In particular negative EGID values are display groups so that they will be ignored in model assembly operations.

The combined models can then be used to create the test/analysis correlation using fe_sens. An application is given in the gartte demo, where a procedure to match initially different test and FE coordinate frames is outlined.

model=feutil('addtest',model1,model2) is a higher level command that attempts to merge two models and retain as much information as possible (nodes, elements, materials, etc.)

Divide *div1 div2 div3*

Mesh refinement by division of elements. Divide applies to all groups in FEel0 (or to the full model with the feutil call). To apply the division to a selection within the model use feutil ObjectDivide. Currently supported divisions are

- segments : elements with beam1 parents are divided in *div1* segments of equal length
- quadrilaterals: elements with quad4 or quadb parents are divided in a regular mesh of *div1* by *div2* quadrilaterals
- hexahedrons: elements with hexa8 or hexa20 parents are divided in a regular grid of *div1* by *div2* by *div3* hexahedrons
- tria3 can be divided with a equal division of each segment specified by div1

If your elements have a different name but the same topological structure declare the proper parent name or use the SetGroupName command before and after divide. The division preserves properties other than the node numbers.

You can obtain unequal divisions by declaring additional arguments whose lines give the relative positions of dividers. For example, an unequal 2 by 3 division of a quad4 element would be obtained using femesh('divide',[0 .1 1],[0 .5 .75 1]) (see also the gartfe demo).

```
% Example 1 : beam1
femesh('reset');
femesh(';testbeam1;divide 3;plotel0'); % divide by 3
fecom textnode

% Example 2 : you may create a command string
number=3;
st=sprintf(';testbeam1;divide %f;plotel0',number);
femesh('reset');
femesh(st);
fecom textnode

% Example 3 : you may use uneven division
femesh('reset');
model=femesh('testquad4'); % one quad4 created
model=feutil('divideelt',model,[0 .1 .2 1],[0 .3 1]);
feplot(model);
```

An inconsistency in division for quad elements was fixed with version 1.105, you can obtain the consistent behaviour (first division along element x by adding -new anywhere in the divide command.

DivideInGroups

Finds groups of FEel0 elements that are not connected (no common node) and places each of these groups in a single element group.

DivideGroup *i ElementSelectors*

Divides a single group *i* of FEelt in two element groups. The first new element group is defined based on the element selectors (see section 5.12).

This function is also accessible using feutil. For example

```
elt=feutil('divide group 1 withnode{x>10}',model)
```

EltId

[EltId]=feutil('eltid',elt) returns the element identifier for each element in elt. It currently does not fill EltId for elements which do not support it.

`[EltId,elt]=feutil('eltidfix',elt)` returns an `elt` where the element identifiers have been made unique.
Command modifier `-elt` can be used to set new `EltId`.

```
model=femesh('testhexa8')
[EltId,model.Elt]=feutil('EltIdFix',model.Elt); % Fix and get EltId
[model.Elt,EltIdPos]=feutil('eltid-elt',model,EltId*18); % Set new EltId
model.Elt(EltIdPos>0,EltIdPos(EltIdPos>0)) % New EltId
```

Extrude *nRep tx ty tz*

Extrusion. Nodes, lines or surfaces that are currently selected (put in `FEel0`) are extruded ***nRep*** times with global translations ***tx ty tz***. Elements with a `mass1` parent are extruded into beams, element with a `beam1` parent are extruded into `quad4` elements, `quad4` are extruded into `hexa8`, and `quadb` are extruded into `hexa20`.

You can create irregular extrusions. For example, `femesh('extrude 0 0 0 1',[0 logspace(-1,1,5)])` will create an exponentially spaced mesh in the z direction. The second `femesh` argument gives the positions of the sections for an axis such that `tx ty tz` is the unit vector.

```
% Example 1 : beam
femesh('reset');
femesh('testbeam1'); % one beam1 created
femesh(';extrude 2 1 0 0;plotel0'); % 2 extrusions in x direction

% Example 2 : you may create the command string
number=2;step=[1 0 0];
st=sprintf(';testbeam1;extrude %f %f %f %f',[number step]);
femesh('reset');
femesh(st);  femesh plotel0

% Example 3 : you may uneven extrusions in z direction
femesh('reset');
femesh('testquad4')
femesh('extrude 0 0 0 1', [0 .1 .2 .5 1]); %
% 0 0 0 1        :  1 extrusion in z direction
% [0 .1 .2 .5 1] :  where extrusions are made
femesh plotel0
```

GetDof, FindDof *ElementSelectors*

The nominal call to get DOFs used by a model is `mdof=feutil('GetDOF',model)`. These calls are performed during assembly phases (`fe_mk`, `fe_load`, ...). This supports elements with variable DOF numbers defined through the element rows or the element property rows. To find DOFs of a part of the model, you should build a sub-model as follows

```
model=femesh;
model.Elt=feutil('selelt Group1:3',model);
mdof=feutil('GetDof',model);
```

Note that node numbers set to zero are ignored by `feutil` to allow elements with variable number of nodes.

`FindElt` *`ElementSelectors`*

Find elements based on a number of selectors described in section 5.12. The calling format is

```
[ind,elt] = femesh('findelt withnode 1:10')
```

where `ind` gives the row numbers of the elements (but not the header rows except for unique superelements which are only associated to a header row) and `elt` the associated element description matrix. `FindEl0` applies to elements in `FEel0`. This command can be accessed directly with `feutil`. The example above is equivalent to

```
[ind,elt]=feutil('findelt eltid 1:10 ',model)
```

When operators are accepted, equality and inequality operators can be used. Thus `group~=[3 7]` or `pro < 5` are acceptable commands. See also `SelElt`, `RemoveElt` and `DivideGroup`, the `gartfe` demo, `fecom` selections.

`FindNode` *`Selectors`*

Find node numbers based on a number of selectors listed in section 5.11.

Different selectors can be chained using the logical operations `&` (finds nodes that verify both conditions), `|` (finds nodes that verify one or both conditions). Condition combinations are always evaluated from left to right (parentheses are not accepted).

Output arguments are the numbers `NodeID` of the selected nodes and the selected nodes `node` as a second optional output argument. This command is equivalent to the `feutil` call

`[NodeID,node]=feutil(['findnode ...'],FEnode, FEelt,FEel0)`.

As a example you can show node numbers on the right half of the `z==0` plane using the commands

```
fecom('TextNode',femesh('findnode z==0 & x>0'))
```

Following example puts markers on selected nodes

```
demosdt('demo ubeam'); cf=feplot; % load U-Beam model
fecom('ShowNodeMark',feutil('FindNode z>1.25',cf.mdl),'color','r')
fecom('ShowNodeMark',feutil('FindNode x>0.2*z|x<-0.2*z',cf.mdl),...
    'color','g','marker','o')
```

Note that you can give numeric arguments to the command as additional `femesh` arguments. Thus the command above could also have been written

```
fecom('TextNode',femesh('findnode z== & x>=',0,0)))
```

See also the `gartfe` demo.

GetEdge[Line,Patch]

These feutil commands are used to create a mode containing the 1D edges or 2D faces of a model. A typical call is

```
femesh('reset');model=femesh('testubeam');
elt=feutil('getedgeline',model);feutil('infoelt',elt)
```

GetEdgeLine supports the following variants MatId retains inter material edges, ProId retains inter property edges, Group retains inter group edges, all does not eliminate internal edges, InNode only retains edges whose node numbers are in a list given as an additional feutil argument.

These commands are used for SelEdge and SelFace element selection commands. Selface preserves the EltId and adds the FaceId after it to allow face set recovery.

GetElemF

Header row parsing. In an element description matrix, element groups are separated by header rows (see section 5.2) which for the current group jGroup is given by elt(EGroup(jGroup),:). The GetElemF command, whose proper calling format is

```
[ElemF,opt,ElemP] = feutil('getelemf',elt(EGroup(jGroup),:),[jGroup])
```

returns the element/superelement name ElemF, element options opt and the parent element name ElemP. It is expected that opt(1) is the EGID (element group identifier) when defined.

GetLine

Line=feutil('get Line',node,elt) returns a matrix of lines where each row has the form [length(ind)+1 ind] plus trailing zeros, and ind gives node indices (if the argument node is not empty) or node numbers (if node is empty). elt can be an element description matrix or a connectivity line matrix (see feplot). Each row of the Line matrix corresponds to an element group or a line of a connectivity line matrix. For element description matrices, redundant lines are eliminated.

GetNode *Selectors*

node=femesh('get node *Selectors*') returns a matrix containing nodes rather than node indices obtained with the feutil FindNode command. This command is equivalent to the feutil call

```
node=feutil(['findnode ...'],FEnode, FEelt,FEel0).
```

GetNormal[Elt,Node][,Map],GetCG

[normal,cg]=feutil('getNormal[elt,node]',node,elt) returns normals to elements/nodes in model node, elt. CG=feutil('GetCg',model) returns the CG locations. MAP=feutil('getNormal Map',model) returns a data structure with the following fields

ID	identifier. One integer per vector in the field map. Typically node numbers or EltId.
vertex	$N \times 3$ matrix giving vertex positions if the map is not associated with nodes
normal	$N \times 3$ where each row specifies a vector at ID or vertex
opt	1 for MAP at element center, 2 for map at nodes.

GetPatch

Patch=feutil('get Patch',node,elt) returns a patch matrix where each row (except the first which serves as a header) has the form [n1 n2 n3 n4 EltN GroupN]. The n*i* give node indices (if the argument node is not empty) or node numbers (if node is empty). elt must be an element description matrix. Internal patches (it is assumed that a patch declared more than once is internal) are eliminated.

Info [,FEel*i*, Node*i*]

Information on global variables. Info by itself gives information on all variables. The additional arguments FEelt ... can be used to specify any of the main t, selected 0 and alternate 1 finite element model description matrices. InfoNode*i* gives information about all elements that are connected to node *i*. To get information in FEelt and in FEnode, you may write

femesh('InfoElt') or femesh('InfoNode')

The equivalent feutil calls would be

feutil('InfoElt',model) or feutil('InfoNode',model)

Join [,el0] [group *i*, *EName*]

Join the groups *i* or all the groups of type *EName*. By default this operation is applied to FEelt but you can apply it to FEel0 by adding the el0 modifier to the command. Note that with the selection by group number, you can only join groups of the same type (with the same element name).

You may join groups using there ID

```
femesh('reset');
femesh(';test2bay;plotelt');
femesh('infoelt');   % 2 groups at this step
femesh joingroup1:2  % 1 group now
```

or using elements type

```
femesh('reset');
femesh('test2bay;plotelt');
femesh joinbeam1     % 1 group now
```

This command can be accessed directly with feutil. For example

```
elt=feutil('joingroup1:2',model.Elt)
```

model [,0]

`model=femesh('model')` returns the FEM structure (see section 5.6) with fields `model.Node=FEnode` and `model.Elt=FEelt` as well as other fields that may be stored in the `FE` variable that is persistent in `femesh`. `model=femesh('model0')` uses `model.Elt=FEel0`.

Matid,ProId,MPID

`[MatId]=feutil('matid',elt)` returns the element material identifier for each element in `elt`. The `ProId` command works similarly. `MPId` returns a matrix with three columns `MatId`, `ProId` and group numbers.
`elt=feutil('mpid',elt,mpid)` can be used to set properties.

ObjectBeamLine *i*, ObjectMass *i*

Create a group of `beam1` *elements.* The node numbers *i* define a series of nodes that form a continuous beam (for discontinuities use `0`), that is placed in `FEel0` as a single group of `beam1` elements.

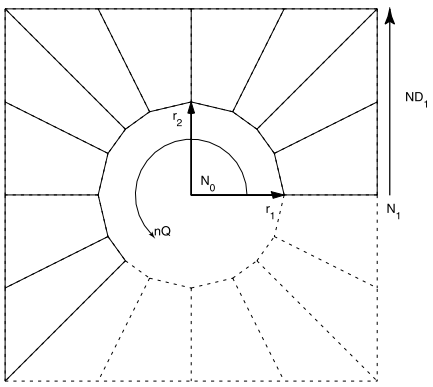
For example `femesh('ObjectBeamLine 1:3 0 4 5')` creates a group of three `beam1` elements between nodes `1 2`, `2 3`, and `4 5`.

An alternate call is `femesh('ObjectBeamLine',ind)` where `ind` is a vector containing the node numbers. You can also specify a element name other than `beam1` and properties to be placed in columns 3 and more using `femesh('ObjectBeamLine -EltName',ind,prop)`.

`femesh('ObjectMass 1:3')` creates a group of concentrated `mass1` elements at the declared nodes.

```
FEnode = [1 0 0 0  0  0 0;   2 0 0 0  0  0 .15; ...
          3 0 0 0 .4  1 .176;4 0 0 0 .4 .9 .176];
femesh(';objectbeamline 1 2 0 2 3 0 3 4');
% or femesh('objectbeamline',1:4);
femesh('object mass',3,[1.1 1.1 1.1])
femesh plotel0;fecom textnode
```

ObjectHoleInPlate



Create a quad4 *mesh of a hole in a plate.* The format is *'ObjectHoleInPlate N0 N1 N2 r1 r2 ND1 ND2 NQ'* giving the center node, two nodes to define the edge direction and distance, two radiuses in the direction of the two edge nodes (for elliptical holes), the number of divisions along a half quadrant of edge 1 and edge 2, the number of quadrants to fill (the figure shows 2.5 quadrants filled).

```
FEnode = [1 0 0 0  0 0 0; 2 0 0 0  1 0 0; 3 0 0 0  0 2 0];
femesh('objectholeinplate 1 2 3 .5 .5 3 4 4');
femesh('divide 3 4'); % 3 divisions around, 4 divisions along radii
femesh plotel0
% You could also use the call
FEnode = [1 0 0 0  0 0 0;  2 0 0 0  1 0 0; 3 0 0 0  0 2 0];
%   n1 n2 n3 r1 r2 nd1 nd2 nq
r1=[ 1  2  3 .5 .5  3   4   4];
st=sprintf(';objectholeinplate %f %f %f %f %f %f %f %f',r1);
femesh(st);femesh('plotel0')
```

Object[Quad,Beam,Hexa] *MatId ProId*

Create or add a model containing quad4 *elements.* The user must define a rectangular domain delimited by four nodes and the division in each direction. The result is a regular mesh.

For example `feutil('ObjectQuad 10 11',nodes,4,2)` returns model with 4 and 2 divisions in each direction with a *MatId* 10 and a *ProId* 11.

An alternate call is `model=feutil('ObjectQuad 1 1',model,nodes,4,2)` : the quadrangular mesh is added to the model.

```
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
model=feutil('Objectquad 1 1',node,4,3); % creates model

node = [3  0  0; 5  0  0; 5  2  0; 3  2  0];
model=feutil('Objectquad 2 3',model,node,3,2); % matid=2, proid=3
feplot(model);
```

Divisions may be specified using a vector between `[0,1]` :

```
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
model=feutil('Objectquad 1 1',node,[0 .2 .6 1],linspace(0,1,10));
feplot(model);
```

Other supported object topologies are beams and hexaedrons. For example

```
node = [0  0  0; 2  0  0;1  3  0; 1  3  1];
model=feutil('Objectbeam 3 10',node(1:2,:),4); % creates model
model=feutil('Objecthexa 4 11',model,node,3,2,5); % creates model
feutil('infoelt',model)
```

`Object[Arc, Annulus, Circle,Cylinder,Disk]`

These object constructors follow the format

```
model=feutil('ObjectAnnulus x y z r1 r2 nx ny nz Nseg NsegR',model)
model=feutil('ObjectArc xc yc zc x1 y1 z1 x2 y2 z2 Nseg obt',model)
model=feutil('ObjectCircle x y z r nx ny nz Nseg',model)
model=feutil('ObjectCylinder x1 y1 z1 x2 y2 z2 r divT divZ',model)
model=feutil('ObjectDisk x y z r nx ny nz Nseg NsegR',model)
model=feutil('object arc 0 0 0 1 0 0 0 1 0 30 1');
model=feutil('object arc 0 0 0 1 0 0 0 1 0 30 1',model);
model=feutil('object circle 1 1 1 2 0 0 1 30',model);
model=feutil('object circle 1 1 3 2 0 0 1 30',model);
model=feutil('object cylinder 0 0 0  0 0 4 2 10 20',model);
model=feutil('object disk 0 0 0 3 0 0 1 10 3',model);
model=feutil('object annulus 0 0 0 2 3 0 0 1 10 3',model);
feplot(model)
```

`ObjectDivide`

Applies a divide command to a selection within the model

```
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
model=feutil('Objectquad 1 1',node,4,3); % creates model
model=feutil('objectDivide 3 2',model,'withnode 1');
feplot(model);
```

`Optim [Model, NodeNum, EltCheck]`

`OptimModel` removes nodes unused in `FEelt` *from* `FEnode`.

`OptimNodeNum` does a permutation of nodes in `FEnode` such that the expected matrix bandwidth is smaller. This is only useful to export models, since here DOF renumbering is performed by `fe_mk`.

`OptimEltCheck` attempts to fix geometry pathologies (warped elements) in `quad4`, `hexa8` and `penta6` elements.

`Orient, Orient` *`i`* `[ , n` *`nx ny nz`*`]`

Orient elements. For volumes and 2-D elements which have a defined orientation. `femesh('orient')` or the equivalent `elt=feutil('orient',FEnode,FEelt)` call element functions with standard material properties to determine negative volume orientation and permute nodes if needed. This is in particular needed when generating

models via extrude or divide operations which do not necessarily result in appropriate orientation (see integrules). When elements are too distorted, you may have a locally negative volume. A warning about warped volumes is then passed. You should then correct your mesh. Note that for 2D meshes you need to use 2D topology holders q4p, t3p, ...

Orient normal of shell elements. For plate/shell elements (elements with parents of type quad4, quadb or tria3) in groups *i* of FEelt, this command computes the local normal and checks whether it is directed towards the node located at *nx ny nz*. If not, the element nodes are permuted to that a proper orientation is achieved. A -neg can be added at the end of the command to force orientation away rather than towards the nearest node.

femesh('orient *i*',node) can also be used to specify a list of orientation nodes. For each element, the closest node in node is then used for the orientation. node can be a standard 7 column node matrix or just have 3 columns with global positions.

For example

```
% Init example
femesh('reset');
femesh(';testquad4;divide 2 3;')
FEelt=FEel0;femesh('dividegroup1 withnode1');
model=femesh;
% Orient elements in group 2 away from [0 0 -1]
model.Elt=feutil('orient 2 n 0 0 -1 -neg',model);
MAP=feutil('getnormal MAP',model);MAP.normal
```

Plot [Elt, El0]

Plot selected model. PlotElt calls feplot to initialize a plot of the model contained in FEelt. PlotEl0 does the same for FEel0. This command is really just the declaration of a new model using feplot('initmodel',femesh('model')).

Once the plot initialized you can modify it using feplot and fecom.

Lin2quad, Quad2Tria, quad42quadb, etc.

Basic element type transformations. Quad2Tria searches FEel0 for quad4 element groups and replaces them with equivalent tria3 element groups. The result is stored in FEel0.

[model.Node,model.Elt]=feutil('lin2quad epsl .01',model); is the generic command to generate second order meshes. Lin2QuadCyl places the mid-nodes on cylindrical arcs. Lin2QuadKnowNew can be used to get much faster results if it is known that none of the new mid-edge nodes is coincident with an existing node.

Obsolete calls are Quad42Quadb places nodes at the mid-sides of quad4 elements to form 8 node quadb elements. Penta62Penta15 (resp. Tetra42Tetra10) transforms penta6(resp.) elements to penta15(resp. tetra10) elements. Hexa82Hexa20 places nodes at the mid-sides of hexa8 elements to form hexa20 elements.

Hexa2Tetra replaces each hexa8 elements by four tetra4 elements (this is really not a smart thing to do). Hexa2Penta replaces each hexa8 elements by six tetra4 elements (warning : this transformation may lead to incompatibilities on the triangular faces).

```
% create 4 quad4
femesh(';testquad4;divide 2 3');
femesh(';quad2tria'); % conversion
femesh plotel0
% create a quad, transform to triangles, divide each triangle in 4
femesh(';testquad4;quad2tria;divide2;plotel0;info');
% example with feutil call  - - - - - - - -  - - -
model=femesh('testhexa8');
[model.Node,model.Elt]=feutil('lin2quad epsl .01',model);
feutil('infoelt',model)
```

RefineBeam *l*

Mesh refinement. This function searches FEel0 for beam elements and divides elements so that no element is longer than *l*.

Remove[Elt,El0] *ElementSelectors*

Element removal. This function searches FEelt or FEel0 for elements which verify certain properties selected by *ElementSelectors* and removes these elements from the model description matrix. The functionality is actually handled by feutil. A sample call would be

```
% create 4 quad4
femesh('reset');
femesh(';testquad4;divide 2 3');
[FEel0,RemovedElt] = feutil('removeelt withnode 1',FEnode,FEel0);
% same as femesh('removeel0 withnode 1')
femesh plotel0
```

Renumber

model=feutil('renumber',model,NewNodeNumbers) can be used to change the node numbers in the model. Currently nodes, elements, DOFs and deformations are renumbered. If NewNodeNumbers is not provided values 1:size(model.Node,1) are used. This command can be used to meet the OpenFEM requirement that node numbers be less than 2^31/100. Another application is to joint disjoint models with coincident nodes using

```
[r1,i2]=feutil('addnode',model.Node,model.Node);
model=feutil('renumber',model,r1(i2,1));
```

RepeatSel *nITE tx ty tz*

Element group translation/duplication. RepeatSel repeats the selected elements (FEel0) *nITE* times with global axis translations *tx ty tz* between each repetition of the group. If needed, new nodes are added to FEnode. An example is treated in the d_truss demo.

```
femesh('reset');
femesh(';testquad4;divide 2 3');
femesh(';repeatsel 3 2 0 0'); % 3 repetitions, translation x=2
femesh plotel0
% an alternate call would be
femesh(';testquad4;divide 2 3');
%                                      number, direction
femesh(sprintf(';repeatsel %f %f %f %f', 3,    [2 0 0]))
femesh plotel0
```

Rev *nDiv OrigID Ang nx ny nz*

Revolution. The selected elements FEel0 are taken to be the first meridian. Other meridians are created by rotating the selected group around an axis passing trough the node of number *OrigID* (or the origin of the global coordinate system) and of direction [*nx ny nz*] (the default is the z axis [0 0 1]). *nDiv+1* (for closed circle cases ang=360, the first and last are the same) meridians are distributed on a sector of angular width *Ang* (in degrees). Meridians are linked by elements in a fashion similar to extrusion. Elements with a mass1 parent are extruded into beams, element with a beam1 parent are extruded into quad4 elements, quad4 are extruded into hexa8, and quadb are extruded into hexa20.

The origin can also be specified by the *xyz* values preceded by an o using a command like femesh('rev 10 o 1.0 0.0 0.0 360 1 0 0').

You can obtain an uneven distribution of angles using a second argument. For example femesh ('rev 0 101 40 0 0 1',[0 .25 .5 1]) will rotate around an axis passing by node 101 in direction *z* and place meridians at angles 0 10 20 and 40 degrees. Note that *SDT 4.0* did not behave correctly for such calls.

```
FEnode = [1 0 0 0  .2 0   0;  2 0 0 0  .5 1 0; ...
          3 0 0 0  .5 1.5 0;4 0 0 0  .3 2 0];
femesh('objectbeamline',1:4);
femesh('divide 3')
femesh(';rev 40 o 0 0 0 360 0 1 0');
femesh plotel0
fecom(';triax;view 3;showpatch')
% An alternate calling format would be
femesh(';objectbeamline 1:4;divide3');
%    divi origin angle direct
r1 = [40  0 0 0  360   0 1 0];
femesh(sprintf(';rev %f o %f %f %f %f %f %f %f',r1))
femesh plotel0
fecom(';triax;view 3;showpatch')
```

RotateSel *OrigID Ang nx ny nz*

Rotation. The selected elements FEel0 are rotated by the angle ***Ang*** (degrees) around an axis passing trough the node of number ***OrigID*** (or the origin of the global coordinate system) and of direction [***nx ny nz***] (the default is the z axis [0 0 1]). The origin can also be specified by the *xyz* values preceded by an o

```
femesh('rotatesel o 2.0 2.0 2.0    90 1 0 0')
```

This is an example of the rotation of FEel0

```
femesh('reset');
femesh(';testquad4;divide 2 3');
% center is node 1, angle 30, aound axis z
%                                        Center angle dir
st=sprintf(';rotatesel %f %f %f %f %f',[1      30   0 0 1]);
femesh(st);  femesh plotel0
fecom(';triax;textnode'); axis on
```

Sel [Elt,El0] *ElementSelectors*

Element selection. SelElt places in the selected model FEel0 elements of FEelt that verify certain conditions. You can also select elements within FEel0 with the SelEl0 command. Available element selection commands are described under the FindElt command and section 5.12.

With arguments use elt=feutil('selelt *ElementSelectors*',model).

SelGroup *i*, SelNode *i*

Element group selection. The element group *i* of FEelt is placed in FEel0 (selected model). SelGroup*i* is equivalent to SelEltGroup*i*.

Node selection. The node(s) *i* of FEnode are placed in FEn0 (selected nodes).

SetGroup [*i*,*name*] [Mat *j*, Pro *k*, EGID *e*, Name *s*]

Set properties of a group. For group(s) of FEelt selector by number *i*, name ***name***, or all you can modify the material property identifier *j*, the element property identifier ***k*** of all elements and/or the element group identifier *e* or name *s*. For example

```
femesh('set group1:3 pro 4')
femesh('set group rigid name celas')
```

If you know the column of a set of element rows that you want to modify, calls of the form FEelt(femesh('findelt*Selectors*'),*Column*)= *Value* can also be used.

```
model=femesh('testubeamplot');
FEelt(femesh('findeltwithnode {x==-.5}'),9)=2;
femesh plotelt;
cf.sel={'groupall','colordatamat'};
```

You can also use femesh('set groupa 1:3 pro 4') to modify properties in FEel0.

StringDOF

`feutil('stringdof',sdof)` returns a cell array with cells containing string descriptions of the DOFs in `sdof`.

SymSel *OrigID nx ny nz*

Plane symmetry. `SymSel` replaces elements in `FEl0` by elements symmetric with respect to a plane going through the node of number *OrigID* (node `0` is taken to be the origin of the global coordinate system) and normal to the vector `[`*nx ny nz*`]`. If needed, new nodes are added to `FEnode`. Related commands are `TransSel`, `RotateSel` and `RepeatSel`.

TransSel *tx ty tz*

Translation of the selected element groups. `TransSel` replaces elements of `FEl0` by their translation of a vector `[`*tx ty tz*`]` (in global coordinates). If needed, new nodes are added to `FEnode`. Related commands are `SymSel`, `RotateSel` and `RepeatSel`.

```
femesh('reset');
femesh(';testquad4;divide 2 3;addsel');
femesh(';transsel 3 1 0;addsel'); % Translation of [3 1 0]
femesh plotelt
fecom(';triax;textnode')
```

UnJoin *Gp1 Gp2*

Duplicate nodes that are common to two groups. To allow the creation of interfaces with partial coupling of nodal degrees of freedom, `UnJoin` determines which nodes are common to the element groups *Gp1* and *Gp2* of `FElt`, duplicates them and changes the node numbers in *Gp2* to correspond to the duplicate set of nodes. In the following call with output arguments, the columns of the matrix `InterNode` give the numbers of the interface nodes in each group `InterNode = femesh('UnJoin 1 2')`.

```
femesh('reset');
femesh('test2bay');
femesh('findnode group1 & group2') % nodes 3 4 are common
femesh('unjoin 1 2');
femesh('findnode group1 & group2') % no longer any common node
```

A more general call allows to separate nodes that are common to two sets of elements `femesh('unjoin','Section1','Selection2')`. Elements in *Selection1* are left unchanged while nodes in *Selection2* that are also in `Selection1` are duplicated.

See also `fe_mk`, `fecom`, `feplot`, section 3.6, demos `gartfe`, `d_ubeam`, `beambar` ...

fe_c

Purpose DOF selection and input/output shape matrix construction.

Syntax

```
c             = fe_c(mdof,adof)
c             = fe_c(mdof,adof,cr,ty)
b             = fe_c(mdof,adof,cr)'
[adof,ind,c] = fe_c(mdof,adof,cr,ty)
ind           = fe_c(mdof,adof,'ind',ty)
adof          = fe_c(mdof,adof,'dof',ty)
labels        = fe_c(mdof,adof,'dofs',ty)
```

Description This function is quite central to the flexibility of DOF numbering in the *Toolbox*. FE model matrices are associated to *DOF definition vectors* which allow arbitrary DOF numbering (see section 5.5). `fe_c` provides simplified ways to extract the indices of particular DOFs (see also section 5.10) and to construct input/output matrices. The input arguments for `fe_c` are

`mdof`	*DOF definition vector* for the matrices of interest (be careful not to mix DOF definition vectors of different models)
`adof`	*active DOF definition vector.*
`cr`	*output matrix associated to the active DOFs.* The default for this argument is the identity matrix. `cr` can be replaced by a string `'ind'` or `'dof'` specifying the unique output argument desired then.
`ty`	*active/fixed option* tells `fe_c` whether the DOFs in `adof` should be kept (`ty=1` which is the default) or on the contrary deleted (`ty=2`).

The input `adof` can be a standard DOF definition vector but can also contain wild cards as follows

`NodeID.0`	means all the DOFs associated to node `NodeID`
`0.DofID`	means `DofID` for all nodes having such a DOF
`-EltID.0`	means all the DOFs associated to element `EltID`

The convention that DOFs `.07` to `.12` are the opposite of DOFs `.01` to `.06` is supported by `fe_c`, but this should really only be used for combining experimental and analytical results where some sensors have been positioned in the negative directions.

The output argument `adof` is the actual list of DOFs selected with the input argument. `fe_c` seeks to preserve the order of DOFs specified in the input `adof`. In particular for models with nodal DOFs only and

- `adof` contains no wild cards: no reordering is performed
- `adof` contains node numbers: the expanded `adof` shows all DOFs of the different nodes in the order given by the wild cards

The first use of `fe_c` is the **extraction** of particular DOFs from a DOF definition vector (see `b,c` page 77). One may for example want to restrict a model to 2-D

motion in the xy plane (impose a fixed boundary condition). This is achieved as follows

```
[adof,ind] = fe_c(mdof,[0.01;0.02;0.06]);
mr = m(ind,ind); kr = k(ind,ind);
```

Note `adof=mdof(ind)`. The vector `adof` is the DOF definition vector linked to the new matrices `kr` and `mr`.

Another usual example is to fix the DOFs associated to particular nodes (to achieve a clamped boundary condition). One can for example fix nodes 1 and 2 as follows

```
ind = fe_c(mdof,[1 2],'ind',2);
mr = m(ind,ind); kr = k(ind,ind);
```

Displacements that do not correspond to DOFs can be fixed using `fe_coor`.

The second use of `fe_c` is the creation of **input/output shape matrices** . These matrices contain the position, direction, and scaling information that describe the linear relation between particular applied forces (displacements) and model coordinates. `fe_c` allows their construction without knowledge of the particular order of DOFs used in any model (this information is contained in the DOF definition vector `mdof`). For example the output shape matrix linked to the relative x translation of nodes 2 and 3 is simply constructed using

```
c=fe_c(mdof,[2.01;3.01],[1 -1])
```

For reciprocal systems, input shape matrices are just the transpose of the collocated output shape matrices so that the same function can be used to build point load patterns.

Example

Others examples may be found in `adof` section.

See also `fe_mk`, `feplot`, `fe_coor`, `fe_load`, `adof`,
Section 3.7

fe_case

Purpose UI function to handle FEM computation *cases*

Syntax

```
Case = fe_case(Case,'EntryType','Entry Name',Data)
fe_case(model,'command' ...)
```

Description *FEM computation cases* contain information other than nodes and elements used to describe a FEM computation. Currently supported entries in the case stack are

cyclic	(SDT) used to support cyclic symmetry conditions
DofLoad	loads defined on DOFs (handled by fe_load)
DofSet	(SDT) imposed displacements on DOFs
FixDof	used to eliminated DOFs specified by the stack data
FSurf	surface load defined on element faces (handled by fe_load). This will be phased out since surface load elements assicated with volume loads entries are more general.
FVol	volume loads defined on elements (handled by fe_load)
info	used to stored non standard entries
KeepDof	(obsolete) used to eliminated DOFs not specified by the stack data. These entries are less general than FixDof and should be avoided.
map	field of normals at nodes
mpc	multiple point constraints
rbe3	a flavor of MPC that enforce motion of a node a weighted average
par	are used to define physical parameters (see upcom par commands
rigid	linear constraints associated with rigid links
SensDof	(SDT) Sensor definitions

fe_case is called by the user to initialize (when Case is not provided as first argument) or modify cases (Case is provided).

Accepted commands are

- Assemble[...] calls used to assemble the matrices of a model. Accepted formats for matrix assembly are

  ```
  [m,k,model,Case]=fe_case(model,'assemble mk');
  [k,model,Case] = fe_case(model,'assemble k');
  [ ... ] = fe_case(model,'assemble ...',Case);
  ```

  Note that constraints are eliminated from the resulting matrices (see section 5.13).

- Auto-SPC analyses the rank of the stiffnes matrix at each node and generates a fixdof case entry for DOFs found to be singular:

  ```
  model = fe_case(model,'autospc')
  ```

- `[Case,CaseName]=fe_case(model,'GetCase')` returns the current case. `GetCase`*i* returns case number *i* (order in the model stack). `GetCase`*Name* returns a case with name `Name` and creates it does not exist necessary. Note that the Case name cannot start with `Case`.
- `data=fe_case(model,'GetData` *EntryName*`')` returns data associated with the case entry *EntryName*.
- `model=fe_case(model,'SetData` *EntryName*`',data)` sets data associated with the case entry *EntryName*.
- `GetT` returns a congruent transformation matrix which verifies constraints. Details are given in sermpc.
- `model=fe_case(model,'Remove',`*EntryName*`)` removes the entry with name *EntryName*.
- `Reset` empties all information in the case stored in a model structure :

```
model = fe_case(model,'reset')
```

Entries The following paragraphs list available entries not handled by `fe_load` or `upcom`.

`cyclic (SDT)`

`cyclic` entries are used to define sector edges for cyclic symmetry computations. They are generated using the `fe_cyclic Build` command.

`FixDof`

`FixDof` entries correspond to rows of the `Case.Stack` cell array giving `{''FixDof'', Name, Data}`. `Name` is a string identifying the entry. `data` is a column DOF definition vector (see section 5.10) or a string defining a node selection command. You can also use
`data=struct('data',DataStringOrDof,'ID',ID)` to specify a identifier.

You can now add DOF and ID specifications to the `findnode` command. For example `x==0 -dof 1 2 -ID 101` fixes DOFs x and y on the `x==0` plane and generates an `data.ID` field equal to 101 (for use in other software).

The following syntax is used in the final example of the section:

```
model = fe_case(model,'FixDof','clamped dofs','z==0');
```

`map`

`map` entries are used to define maps for normals at nodes. These entries are typically used by shell elements or by meshing tools. `Data` is a structure with fields

- `.normal` a N by 3 matrix giving the normal at each node or element

- .ID a N by 1 vector giving identifiers. For normals at integration points, element coordinates can be given as two or three additional columns.
- .opt an option vector. opt(1) gives the type of map (1 for normals at element centers, 2 for normals at nodes, 3 normals at integration points specified as additional columns of Data.ID).
- .vertex an optional N by 3 matrix giving the location of each vector specified in .normal. This can be used for plotting.

MPC

MPC (multiple point constraint) entries are rows of the Case.Stack cell array giving {'MPC', Name, Data}. Name is a string identifying the entry. Data is a structure with fields Data.ID positive integer for identification. Data.c is a sparse matrix whose columns correspond to DOFs in Data.DOF. c is the constraint matrix such that $[c]\{q\} = \{0\}$ for q defined on DOF.

Data.slave is an optional vector of slave DOFs in Data.DOF. If the vector does not exist, it is filled by feutil FixMpcMaster.

Note that the current implementation has no provision for using local coordinates in the definition of MPC (they are assumed to be defined using global coordinates).

par (SDT)

Parameter entries are used to define variable coefficients in element selections. It is nominally used through upcom Par commands but other routines may also use it [**?**].

RBE3 (SDT)

rbe3 constraints enforce the motion of a slave node as a weighted average of master nodes. The each row of data.data codes an set of constraints following the format

Rbe3ID NodeIdSlave DofSlave Weight1 DofMaster1 NodeId1 ...

DofMaster and DofSlave code which DOFs are used (123 for translations, 123456 for both translations and rotations). There are known robustness problems with the current implementation of this constraint.

rigid

See details under rigid which also illustrates the RigidAppend command.

Sens ... (SDT)

SensDof entries are detailed in section **??**. They are stored as rows of the Case.Stack cell array giving {'SensDof', Name, data}. SensStrain entries have been replaced with strain sensors in SensDof.

un=0

`model=fe_case(model,'un=0','Normal motion',map);` where `map` gives normals at nodes generates an `mpc` case entry that enforces the condition $\{u\}^T\{n\} = 0$ at each node of the map.

Example

Here is an example combining various `fe_case` commands

```
femesh('reset');
model = femesh('test ubeam plot');
% specifying clamped dofs (FixDof)
model = fe_case(model,'FixDof','clamped dofs','z==0');
% creating a volume load
data  = struct('sel','GroupAll','dir',[1 0 0]);
model = fe_case(model,'FVol','Volumic load',data);
% assemble active DOFs and matrices
model=fe_mknl(model);
% assemble RHS (volumic load)
Load  = fe_load(model,'Case1');
% compute static response
kd=ofact(model.K{2});def.def= kd\Load.def; ofact('clear',kd)
Case=fe_case(model,'gett'); def.DOF=Case.DOF;
% plot displacements
feplot('initdef',def);
fecom(';undef;triax;showpatch;promodelinit');
```

See also `fe_mk`, `fe_case`

fe_curve

Purpose Generic handling of curves and signal processing utilities

Syntax

```
out=fe_curve('command',MODEL,'Name',...);
```

Commands

`fe_curve` is used to handle curves and do some basic signal processing. The format for curves is described in section 5.9. The `iiplot` interface may be used to plot curves and a basic call would be `iiplot(Curve)` to plot curve data structure `Curve`.

Accepted commands are

bandpass *Unit f_min f_max*

`out=fe_curve('BandPass Unit f_min f_max',signals);`
realizes a true bandpass filtering (i.e. using `fft()` and `ifft()`) of time signals contained in curves `signals`. `f_min` and `f_max` are given in units `Unit`, whether Hertz(`Hz`) or Radian(`Rd`). With no `Unit`, `f_min` and `f_max` are assumed to be in Hertz.

```
out=fe_curve('TestFrame');% 3 DOF oscillator response to noisy input
fe_curve('Plot',out{2});  % "unfiltered" response
filt_disp=fe_curve('BandPass Hz 70 90',out{2}); % filtering
fe_curve('Plot',filt_disp); title('filtered displacement');
```

datatype [,cell]

`out=fe_curve('DataType',DesiredType);`
returns a data structure describing the data type, usefull to fill `.xunit` and `.yunit` fields for curves definition. `DesiredType` could be a string or a number corresponding to the desired type. With no `DesiredType`, the current list of available types is displayed.

`DataTypeCell` returns a cell array rather than data structure to follow the specification for curve data structures.

getcurve

`curve=fe_curve('getcurve',model,curve_name);`
extracts curve `curve_name` from the `.Stack` field of `model` or the possible curves attached to a load case. If the user does not specify any name, all the curves are returned in a cell array.

h1h2 *input_channels*

`FRF=fe_curve('H1H2 input_channels',frames,window);`
computes H1 and H2 FRF estimators along with the coherence from time signals

contained in cell array `frames` using window `window`. The time vector is given in `frames1.X` while *`input_channels`* tells which colums of in `frames1.Y` are inputs. If more than one input channel is specified, true MIMO FRF estimation is done, and Hν is used instead of H2. When multiple frames are given, a mean estimation of FRF is computed.

Note: To ensure the proper assembly of H1 and Hν in MIMO FRF estimation case, a weighing based on maximum time signals amplitude is used. To use your own, use
`FRF=fe_curve('H1H2 input_channels',frames,window,weighing);`
where `weighing` is a vector containing weighing factors for each channel. To avoid weighing, use
`FRF=fe_curve('H1H2 input_channels',frames,window,0);`

noise

`noise=fe_curve('Noise',Nw_pt,fs,f_max);`
computes a `Nw_pt` points long time signal corresponding to a "white noise", with sample frequency `fs` and a unitary power spectrum density untill `f_max`. `fs/2` is taken as `f_max` when not specified. The general shape of noise power spectrum density, extending from `0` to `fs/2`, can be specified instead of `f_max`.

```
% compute a 2 seconds long white noise, 1024 Hz of sampling freq.
% with "rounded" shape PSD
fs=1024; sample_length=2;
Shape=exp(fe_curve('window 1024 hanning'))-1;
noise_h=fe_curve('noise',fs*sample_length,fs,Shape);
figure(1); subplot(211); % plot time and frequency signals
plot(noise_h.X,noise_h.Y);axis([0 2 -3 3]); xlabel('Time');
subplot(212);
freq=fs*[0:length(noise_h.X)-1]/length(noise_h.X);
plot(freq,20*log10(abs(fft(noise_h.Y))));
axis([0 1024 -20 40]); xlabel('Frequency');
```

plot

`fe_curve('plot',curve);`
plots the curve `curve` named `curve_name`.
`fe_curve('plot',fig_handle,curve);`
plots curve in the figure with handle `fig_handle`.
`fe_curve('plot',model,curve_name);`
`fe_curve('plot',fig_handle,model,curve_name);`
plots curve named `curve_name` stacked in `.Stack` field of model `model`.

```
% compute a 2 seconds long white noise, 1024 Hz of sampling freq.
fs=1024; sample_length=2;
noise=fe_curve('noise',fs*sample_length,fs);
noise.xunit=fe_curve('DataType','Time');
noise.yunit=fe_curve('DataType','Excit. force');
noise.name='Input force';
```

```
fe_curve('Plot',noise);
```

resspectrum [*True, Pseudo*] [*Abs., Rel.*] [*Disp., Vel., Acc.*]

`out=fe_curve('ResSpectrum [T, P] [A, R] [D, V, A]',signal,freq,damp);` computes [*true, pseudo*] [*absolute, relative*] [*displacement, velocitiy, acceleration*] response spectrum associated to the time signal given in `signal`. `signal` is a curve type structure where `.X`, `.Y`, `.ylabel.unit` fields must be filled. `freq` and `damp` are frequencies (in Hz) and damping ratios vectors of interess for the response spectra.

```
st=sprintf('read %s',which('bagnol_ns.cyt'));
bagnol_ns=fe_curve(st); % read the acceleration time signal

st=sprintf('read %s',which('bagnol_ns_rspec_pa.cyt'));
bagnol_ns_rspec_pa= fe_curve(st); % read reference spectrum

% compute response spectrum with reference spectrum frequencies
% vector and 5% damping
RespSpec=fe_curve('ResSpectrum True Rel. Acc.',...
                  bagnol_ns,bagnol_ns_rspec_pa.X/2/pi,.05);

fe_curve('plot',RespSpec); hold on;
plot(RespSpec.X,bagnol_ns_rspec_pa.Y,'r');
legend('fe\_curve','cyberquake');
plot(RespSpec.X,bagnol_ns_rspec_pa.Y,'r');
```

returny

If curve has a `.Interp` field, this interpolation is taking in account. If `.Interp` field is not present or empty, it uses a degree 2 interpolation by default.

To force a specific interpolation (over passing `.interp field`, one may insert the `-linear`, `-log` or `-stair` string in the command.

To extract a curve `,curve_name` and return the values `Y` corresponding to the input `X`, the syntax is

```
y = fe_curve('returny',model,curve_name,X);
```

Given a `curve` data structure to return the values `Y` corresponding to the input `X`, the syntax is

```
y = fe_curve('returny',curve,X);
```

testframe

`out=fe_curve('TestFrame');`
computes the time response of a 3 DOF oscillator to a white noise and fills the cell

array `out` with noise signal in cell 1 and time response in cell 2. It illustrates the use of various functionalities of `fe_curve` and provides typical exemple of curves.

```
fs=512;  ech_length=4;   % sampling frequency and sample length (s)
noise=fe_curve('Noise',fs*ech_length,fs);  % computes noise

% build the curve associated to the time signal of noise
out{1}=struct('X',noise.X,'Y',noise.Y,'xunit',...
       fe_curve('DataType','Time'),'yunit',...
       fe_curve('DataType','Excit. force'),'name','Input at DOF 2');
% set up an oscillator with 3 DOF %
Puls = [30 80 150]'*2*pi;      % natural frequencies
Damp = [.02 .015 .01]';        % damping
Amp = [1 2 -1;2 -1 1;-1 1 2];  % pseudo "mode shapes"
Amp=Amp./det(Amp);

C=[1 0 0]; B=[0 1 0]';   % Observation matrix and Command matrix
freq=([0:length(noise.X)-1]/length(noise.X))*fs*2*pi; % Freq vector

% Eliminating frequencies corresponding to the aliased part
% of the noise spectrum
freq=freq(1:length(noise.X)/2)';
FRF=nor2xf(Puls,Damp,Amp*B,C*Amp,freq); % Transfert function

% Compute the time response to input noise
Resp=fe_curve('TimeFreq',noise,struct('w',freq,'xf',FRF));

% build the curve associated to the time signal of response
out{2}=struct('X',Resp.X,'Y',Resp.Y,'xunit',...
        fe_curve('DataType','Time'),'yunit',...
        fe_curve('DataType','Displacement'),'name','Output at DOF 1');
```

`set`

This command set a curve in the model. 3 types of input are allowed :

- A data structure, `model=fe_curve(model,'set',curve_name,data_structure)`
- A string to interprete, `model=fe_curve(model,'set',curve_name,string)`
- A name refering to an existing curve (for load case only), `model=fe_curve( model, 'set LoadCurve',load_case,chanel,curve_name)`

The following example illustrates the different calls.

```
model=fe_time('demo bar'); q0=[];

% curve defined by a by-hand data structure :
c1=struct('ID',1,'X',linspace(0,1e-3,100), ...
```

```
      'Y',linspace(0,1e-3,100),'data',[],...
      'xunit',[],'yunit',[],'unit',[],'name','curve 1');
model=fe_curve(model,'set','curve 1',c1);
% curve defined by : string to interprete :
model=fe_curve(model,'set','step 1','step 1e-4*10');
% curve defined by a reference curve :
c2=fe_curve('test -ID 100 ricker 10e-4 100 1 100e-4');
c3=fe_curve('test sin 10e-4 100 1 100e-4');

model=fe_curve(model,'set','ricker 1',c2);
model=fe_curve(model,'set','sin 1',c3);

% define Load with curve definition
LoadCase=struct('DOF',[1.01;2.01],'def',1e6*eye(2),...
            'curve',{{fe_curve('test ricker 20e-4 100 1 100e-4'),...
                      'ricker 1'}});
model = fe_case(model,'DOFLoad','Point load 1',LoadCase);
% modify a curve in the load case
model=fe_curve(model,'set LoadCurve','Point load 1',2,'step 1e-4*10');
```

`testFunc`

This command creates curves based on trigonometric and exponential functions; the syntax is

`out=fe_curve(['Test' st],TimeVector);`

where `st=sin, cos, tan, exp`. The `TimeVector` contains the sampling time step, for example: `TimeVector=linspace(0.,1.,100)`.

You can use following format to define period, amplitude and optionally a stop time: `out=fe_curve('Test sin Period Amplitude -stoptime Tf',TimeVector)`.

Note that you can use a command string following the format `out=fe_curve('Test sin Period N Amplitude TotalTime -stoptime Tf')` without defining explicitly a time vector. `N` is the length of the time vector and `Tf` is the optional time to stop the signal (function is 0 after this time).

Without output argument the curve is simply plotted.

`test[Ramp,Ricker]`

`out=fe_curve('TestRamp NStep FinalValue',TimeVector)` generates a ramp composed of `NStep` steps from 0 to `FinalValue`.

`out=fe_curve('TestRicker Duration Amplitude',TimeVector)` generates Ricker functions representing impacts.

`TimeVector` is optional.

For example:

`C1=fe_curve('test ramp 20 2.3',linspace(0,1.2,20));`

```
C2=fe_curve('TestRicker .6 2',linspace(0,1.2,120));
figure(1);plot(C1.X,C1.Y,'-',C2.X,C2.Y,'--')
```

testSweep

`out=fe_curve('TestSweep f0 f1 t0 t1')` generates a sweep cosine from `t0` to `t1`, with linear frequency sweeping from `f0` to `f1`.

$Y = cos(2 * pi * (f0 + \frac{f1-f0}{NStep} * t) * t)$

Note that f1 is not the final instant frequency of the signal (which will be $f0+2*\frac{f1-f0}{NStep}$ for ascending sweep).

testStep

`out=fe_curve('TestStep t1')` generates a step which value is one from time 0 to time `t1`.

testEval

`out=fe_curve('TestEval str')` generates the signal obtained by evaluating the string `str` function of `t`.

For example, one can generate a sine of 150 Hz and amplitude 2 with

```
C1=fe_curve('testEval 2*sin(2*pi*150*t)',linspace(0,1.2,3000));
figure;plot(C1.X,C1.Y);
```

timefreq

`out=fe_curve('TimeFreq',Input,xf);`
computes reponse of a system with given tranfert functions `FRF` to time input `Input`. Sampling frequency and length of time signal `Input` must be coherent with frequency step and length of given transfert `FRF`.

```
fs=1024; sample_length=2;                    % 2 sec. long white noise
noise=fe_curve('noise',fs*sample_length,fs);% 1024 Hz of sampling freq.

w=2*pi*fs*[0:length(noise.X)-1]/length(noise.X); % frequency range
% FRF with resonnant freq. 50 100 200 Hz, unit amplitude, 2% damping
xf=nor2xf(2*pi*[50 100 200].',.02,[1 ; 1 ; 1],[1 1 1],w);

Resp=fe_curve('TimeFreq',noise,xf); % Response to noisy input
fe_curve('Plot',Resp); title('Time response');
```

Window *Nb_pts* [*None, Hanning, Hamming, Exponential*] *Arg*

`win=fe_curve('Window Nb_pts Type Arg');`
computes *Nb_pts* points window. *Arg* is used when *Exponential* window type is asked.

fe_curve

`win = fe_curve('Window 1024 Exponential 10 20 10');` returns an exponential window with 10 zero points, a 20 point flat top, and a decaying exponential over the 1004 remaining points with a last point at `exp(-10)`.

`win = fe_curve('Window 1024 Hanning');` returns a 1024 point long hanning window.

See also `fe_load`, `fe_case`

fe_eig

Purpose Computation of normal modes associated to a second order undamped model.

Syntax

```
[phi, wj] = fe_eig(m,k)
[phi, wj] = fe_eig(m,k,opt,imode)
```

Description The non current *SDT 3* version of this function is included in OpenFEM.

The normal modeshapes `phi`=ϕ and frequencies `wj= sqrt(diag(`Ω^2`))` are solution of the undamped eigenvalue problem

$$-[M]\{\phi_j\}\omega_j^2 + [K]\{\phi_j\} = \{0\}$$

and verify the two orthogonality conditions

$$[\phi]^T[M]_{N\times N}[\phi]_{N\times N} = [I]_{N\times N} \text{ and } [\phi]^T[K][\phi] = \left[\backslash\Omega_j^2{}_{\backslash}\right]$$

Outputs are modeshapes (columns of `phi`) and frequencies `wj` in **rad/s**.

`fe_eig` implements various algorithms to solve this problem for modes and frequencies. Many options are available and it is important that you read the notes below to understand how to properly use them. The format of the option vector `opt` is

`[method nm Shift Print Thres]` (default values are `[2 0 0 0 1e-5]`)

`method`
0 alternate full matrix solution (old method of *SDT* 1.0)
1 subspace iteration which allows to compute the lowest modes of a large problem where sparse mass and stiffness matrices are used. For cases with more than 1000 DOF, the `ofact` object is used to factor the stiffness.
2 default full matrix solution
3 Lanczos algorithm which allows to compute the lowest modes of a large problem using an unpivoted `lu` decomposition of the stiffness matrix
4 same as method 3 but uses a Cholesky decomposition of the stiffness matrix (less general).
...

fe_eig

`method`	**1** subspace iteration is the only partial solver included in this version of the `fe_eig` developped by SDTools. **2** is a FULL solver cleaning up results of the MATLAB `eig` function.
`nm`	number of modes to be returned. A non-integer or negative `nm`, is used as the desired `fmax` in **Hz** for iterative solvers.
`shift`	value of mass shift (should be non-zero for systems with **rigid body modes**, see notes below). The subspace iteration method supports iterations without mass shift for structures with rigid body modes. This method is used by setting the shift value to `Inf`.
`print`	level of printout (`0` none, `11` maximum)
`thres`	threshold for convergence of modes (default `1e-5` for the subspace iteration method)

Notes

- For systems with rigid body modes, you must specify a mass-shift. A good value is about one tenth of the first flexible frequency squared, but the Lanczos algorithm tends to be sensitive to this value (you may occasionaly need to play around a little). If you do not find the expected number of rigid body modes, this is often the reason.
- Memory usage may be affected by the choice of a `skyline method`.

See also `fe_ceig`, `fe_mk`

fe_gmsh

Purpose Information about GMSH can be found at http://www.geuz.org/gmsh/

To call the GMSH mesher from SDT.

Syntax

```
model=fe_gmsh(command,model,...);
```

Description The main operation is the automatic meshing of surfaces.

Example

This example illustrates the automatic meshing of a plate

```
FEnode = [1 0 0 0  0 0 0; 2 0 0 0  1 0 0; 3 0 0 0  0 2 0];
femesh('objectholeinplate 1 2 3 .5 .5 3 4 4');
model=femesh('model0');
model.Elt=feutil('selelt seledge ',model);
model.Node=feutil('getnode groupall',model);
model=fe_gmsh('addline',model,'groupall');
model.Node(:,4)=0; % reset default length
mo1=fe_gmsh('write temp.msh -lc .3 -run -2 -v 0',model);delete('temp.msh')
```

This other example makes a circular hole in a plate

```
% Hole in plate :
model=feutil('Objectquad 1 1',[0 0 0; 1 0 0;1 1 0;0 1 0],1,1); %
model=fe_gmsh('addline',model,[1 2; 2 4; 4 3; 3 1]);
model=fe_gmsh('AddFullCircle',model,[.5 .5 0; .4 .5 0; 0 0 1]);

model.Stack{3}.LineLoop={'Line',[1 2 3 4];'Circle',[1 2 3 4]};
model.Stack{3}.PlaneSurface=[1 2];

mo2=fe_gmsh('write holeinplate.geo -lc .02 -run -2 -v 0',model)
feplot(mo2)
```

To allow automated running of GMSH from MATLAB, this function uses a `info,GMSH` stack entry with the following fields

.Line	one line per row referencing NodeId. Can be defined using addline commands.
.Circle	define properties of circles
.LineLoop	rows define a closed line as combination of elementary lines. Values are row indices in the .Line field. One can also define LineLoop from circle arcs (or mixed arcs and lines) using a cell array whose each row describes a lineloop as {'*LineType*',LineInd,...} where *LineType* can be Circle or Line and LineInd row indices in corresponding .Line or .Circle field.
.PlaneSurface	rows define surfaces as a combination of line loops, values are row indices in the .LineLoop field. Negative values are used to reverse the line orientation.
.SurfaceLoop	rows define a closed surface as combination of elementary surfaces. Values are row indices in the .PlaneSurface field.

The local mesh size is defined at nodes by GMSH. This is stored in column 4 of the model.Node. The -lc .3 in the command resets the value for all nodes that do not have a prior value.

Add...

mdl=fe_gmsh('AddFullCircle',mdl,data) can be used to add to geometry in mdl the circle defined in data. First row of data is the center coordinates, second row is an edge node coordinates and the third row is the normal. 4 arcs of circle are added. One can then define the full circle as a row in the LineLoop field with {'Circle',[ind1 ind2 ind3 ind4]} where indi are the row indices of the 4 arcs of circle created in .Circle field.

mdl=fe_gmsh('AddDisk',mdl,data) ...

mdl=fe_gmsh('AddLine',mdl,data) can be used to add to geometry in mdl the lines defined in data. data can be a 2 column matrix whose each row defines a couple of points from their NodeId. data can also be a 2 by 3 matrix defining the 2 extremities coordinates. data can also be a string defining a line selection.

config

The fe_gmsh function uses the OpenFEM preference to launch the GMSH mesher.

```
setpref('OpenFEM','gmsh','$HOME_GMSH/gmsh.exe')
```

Read

fe_gmsh('read FileName.msh') reads a mesh from the GMSH output format.

See also missread

fe_load

Purpose Interface for the assembly of distributed and multiple load patterns

Syntax

```
Load = fe_load(model)
Load = fe_load(model,Case)
Load = fe_load(model,'NoT')
Load = fe_load(model,Case,'NoT')
```

Description `fe_load` is used to assemble loads (left hand side vectors to FEM problems). Simple point loads are easily built using `fe_c` and reciprocity (transpose of output shape matrix) but `fe_load` is needed for more complex cases.

Loads are associated with cases which are structures with at least `Case.DOF` and `Case.Stack` fields.

```
Case1.DOF = model.DOF; % default is model.DOF
Case1.Stack = [{'LoadType','Name',TypeSpecificData}];
```

Taking the example of a point load with type specific data given by

```
data=struct('DOF',365.03,'def',1);
```

you can create a case using low level commands

```
Case1=struct('DOF',model.DOF,'Stack',{{'DofLoad','PointLoad',data}});
```

or with the easier case creation format (using *SDT* function `fe_case`)

```
Case1=fe_case('DofLoad','PointLoad',data);
```

or add a new load to a case defined in the `model.Stack` field

```
model=fe_case(model,'DofLoad','PointLoad',data);
```

To compute the load, the model (a structure with fields `.Node`, `.Elt`, `.pl`, `.il`) must generally be provided with the syntax `Load=fe_load(model,Case)`. If the case is not provided, `fe_load` uses the first case in `model.Stack`.

The optional `'NoT'` argument is used to require loads defined on the full list of DOFs rather than after constraint eliminations computed using `Case.T'*Load.def`.

The rest of this manual section describes supported load types and the associated type specific data.

DofLoad, DOFSet

Loads at DOFs and `DofLoad` *and prescribed displacements* `DofSet` entries are described by the following data structure

`data.name` name of the case
`data.DOF` column vector containing a DOF selection
`data.def` matrix of load/set for each DOF (each column is a load/set case and the rows are indexed by `Case.DOF`). With two DOFs, `def=[1;1]` is a single intput at two DOFs, while `def=eye(2)` corresponds to two inputs.
`data.lab` cell array giving label, unit label , and unit info (see `fe_curve DataType`) for each load (column of `data.def`)
`data.curve` can specify a curve data structure (or a string referring to an existing curve) to describe frequency or time dependence of loads. Units for the load are defined through the `.lab` field (in $\{F\} = [B]\{u\}$ one assumes u to be unitless thus F and B have the same unit systems).

```
femesh('reset');
model = femesh('testubeam plot');
data=struct('DOF',365.03,'def',1.1); % 1.1 N at node 365 direction z
data.lab=fe_curve('datatype',13);
model=fe_case(model,'DofLoad','PointLoad',data);
% alternate format to declare unit inputs
model=fe_case(model,'DofLoad','ShortTwoInputs',[362.01;258.02]);
Load = fe_load(model);
feplot(model,Load); fecom(';scaleone;undefline;ch1 2') % display
```

`FVol`

`FVol` entries use `data` is a structure with fields

`data.sel` an element selection (or amodel description matrix but this is not acceptable for non-linear applications).
`data.dir` a 3 by 1 cell array specifying the value in each global direction x, y, z. Alternatives for this specification are detailed below . The field can also be specified using `.def` and `.DOF` fields.
`data.lab` cell array giving label, unit label , and unit info (see `fe_curve DataType`) for each load (column of `data.def`)
`data.curve` can specify a curve data structure (or a string referring to an existing curve) to describe frequency or time dependence of loads. Units for the load are defined through the `.lab` field (in $\{F\} = [B]\{u\}$ one assumes u to be unitless thus F and B have the same unit systems).

Each cell of `Case.Dir` can give a constant value (for example gravity), a position dependent value defined by a string `FcnName` that is evaluated using `fv(:,jDir)=eval(FcnName)` or `fv(:,jDir)=feval(FcnName,node)` if the first fails. Note that `node` corresponds to nodes of the model in the global coordinate system.

For example

```
femesh('reset');
model = femesh('testubeam');
data=struct('sel','groupall','dir',[0 9.81 0]);
data2=struct('sel','groupall','dir',{{0,0,'node(:,7)'}});
model=fe_case(model,'FVol','Gravity',data, ...
```

```
                    'FVol','Fv=[0 0 z]',data2);
Load = fe_load(model); feplot(model,Load); % display
```

Note that `feutil('mode2dof')` provides translation to this format from a function defined at nodes.

Volume loads are implemented for all elements, you can always get an example using the elements self tests, for example `[model,Load]=beam1('testload')`.

FSurf

FSurf entries use data a structure with fields

data.sel — a vector of NodeId in which the faces are contained (all the nodes in a loaded face/edge must be contained in the list). data.sel can also contain any valid node selection (using string or cell array format).
the optional data.eltsel field can be used for an optional element selection to be performed before selection of faces with feutil('selelt innode',model,data.sel). The surface is obtained using

```
if isfield(data,'eltsel');
 mo1.Elt=feutil('selelt',mo1,data.eltsel);
end
elt=feutil('seleltinnode',mo1, ...
    feutil('findnode',mo1,r1.sel));
```

data.set — Alternative specification of the loaded face by specifying a face set name to be found in model.Stack

data.def — a vector with as many rows as data.DOF specifying a value for each DOF.

data.DOF — DOF definition vector specifying what DOFs are loaded. Note that pressure is DOF .19. Uniform pressure can be defined using wild cards as show in the example below.

data.lab — cell array giving label, unit label ,and unit info (see fe_curve DataType) for each load (column of data.def)

data.curve — can specify a curve data structure (or a string referring to an existing curve) to describe frequency or time dependence of loads. Units for the load are defined through the .lab field (in $\{F\} = [B]\{u\}$ one assumes u to be unitless thus F and B have the same unit systems).

Surface loads are defined by surface selection and a field defined at nodes. The surface can be defined by a set of nodes (data.sel and possibly data.eltsel fields. One then retains faces or edges that are fully contained in the specified set of nodes. For example

```
femesh('reset');
model = femesh('testubeam plot');
data=struct('sel','x==-.5', ...
            'eltsel','withnode {z>1.25}','def',1,'DOF',.19);
model=fe_case(model,'Fsurf','Surface load',data);
Load = fe_load(model); feplot(model,Load);
```

Or an alternative call with the cell array format for data.sel

```
data=struct('eltsel','withnode {z>1.25}','def',1,'DOF',.19);
```

```
NodeList=feutil('findnode x==-.5',model);
data.sel={'','NodeId','==',NodeList};
model=fe_case(model,'Fsurf','Surface load',data);
Load = fe_load(model); feplot(model,Load);
```

Alternatively, one can specify the surface by referring to a `set` entry in `model.Stack`, as shown in the following example

```
femesh('reset');
model = femesh('testubeam plot');

% Define a face set
[eltid,model.Elt]=feutil('eltidfix',model);
i1=feutil('findelt withnode {x==-.5 & y<0}',model);i1=eltid(i1);
i1(:,2)=2; % fourth face is loaded
data=struct('ID',1,'data',i1);
model=stack_set(model,'set','Face 1',data);

% define a load on face 1
data=struct('set','Face 1','def',1,'DOF',.19);
model=fe_case(model,'Fsurf','Surface load',data);
Load = fe_load(model); feplot(model,Load)
```

The current trend of development is to consider surface loads as surface elements and transform the case entry to a volume load on a surface.

See also `fe_c`, `fe_case`, `fe_mk`

fe_mat

Purpose Material / element property handling utilities.

Syntax

```
out = fe_mat('convert si ba',pl);
typ=fe_mat('m_function',UnitCode,SubType)
[m_function',UnitCode,SubType]=fe_mat('type',typ)
out = fe_mat('unit')
out = fe_mat('unitlabel',UnitSystemCode)
[o1,o2,o3]=fe_mat(ElemP,ID,pl,il)
```

Description Material definitions can be handled graphically using the `Material` tab in the model editor (see section 3.6.4). For general information about material properties, you should refer to section 5.3. For information about element properties, you should refer to section 5.4.

The main user accessible commands in `fe_mat` are listed below

`Convert,Unit [ ,label]`

The `convert` command supports unit conversions to `unit1` to `unit2` with the general syntax

```
pl_converted = fe_mat('convert unit1 unit2',pl);
```

For example convert from SI to BA and back

```
mat = m_elastic('default')
% convert mat.pl from SI unit to BA unit
pl=fe_mat('convert si ba',mat.pl);
% check that conversion is OK
pl2=fe_mat('convert ba si',pl);
mat.pl-pl2(1:6)
```

`out=fe_mat('unitsystem')` returns a `struct` containing the information characterizing standardized unit systems supported in the universal file format.

Code	Identifier	Length and Force
1	SI	Meter, Newton
2	BG	Foot, Pound f
3	MG	Meter, kilogram f
4	BA	Foot, poundal
5	MM	Millimeter, milli-newton
6	CM	Centimeter, centi-newton
7	IN	Inch, Pound force
8	GM	Millimeter, kilogram force
9	TM	Millimeter, Newton
9	US	User defined

Unit codes 1-8 are defined in the universal file format specification and thus coded in the material/element property type (column 2). Other unit systems are considered

user types and are associated with unit code 9. With a unit code 9, `fe_mat convert` commands must give both the initial and final unit systems.

`out=fe_mat('unitlabel',UnitSystemCode)` returns a standardized list of unit labels corresponding in the unit system selected by the `UnitSystemCode` shown in the table above.

When defining your own properties you only need to implement the `p_fun PropertyUnitType` command to allow support of unit conversion.

Get[pl,il]

`pl = fe_mat('getpl',model)` is used to robustly return the material property matrix `pl` (see section 5.3) independently of the material input format.

Similarly `il = fe_mat('getil',model)` returns the element property matrix `il`.

Type

The type of a material or element declaration defines the function used to handle it.

`typ=fe_mat('m_function',UnitCode,SubType)` returns a real number which codes the material function, unit and sub-type. Material functions are `.m` or `.mex` files whose name starts with `m_` and provide a number of standardized services as described in the `m_elastic` reference.

The `UnitCode` is a number between 1 and 9 giving the unit selected. The `SubType` is a also a number between 1 and 9 allowing selection of material subtypes within the same material function (for example, `m_elastic` supports subtypes : 1 isotropic solid, 2 fluid, 3 anisotropic solid).

Note : the code type `typ` should be stored in column 2 of material property rows (see section 5.3).

`[m_function,UnitCode,SubType]=fe_mat('typem',typ)`

Similarly, element properties are handled by `p_` functions which also use `fe_mat` to code the type (see `p_beam`,`p_shell` and `p_solid`).

ElemP

Calls of the form `[o1,o2,o3]=fe_mat(ElemP,ID,pl,il)` are used by element functions to request constitutive matrices. This call is really for developpers only and you should look at the source code of each element.

See also `m_elastic`, `p_shell`, element functions in chapter 6

fe_mk, fe_mknl

Purpose

Assembly of finite element model matrices.

Syntax

```
[m,k,mdof] = fe_mknl(model);
model      = fe_mk(model,'Options',Opt)
[m,k,mdof] = fe_mk( ... ,[0       OtherOptions])
[mat,mdof] = fe_mk( ... ,[MatType OtherOptions])
[Case,model.DOF]=fe_mknl('init',model);
mat=fe_mknl('assemble',model,Case,def,MatType);
```

Description

`fe_mk` and `fe_mknl` take models and return assembled matrices and/or right hand side vectors. `fe_mknl` is the most efficient but has some limitations in the support of superelements. **It should be used by default**.

Input arguments are

- `model` a model data structure describing nodes, elements, material properties, element properties, and possibly a case.
- `Case` a data structure describing loads, boundary conditions, etc. This may be stored in the model and be retrieved automatically using `fe_case(model,'GetCase')`.
- `def` a data structure describing the current state of the model for model/residual assembly using `fe_mknl`. `def` is expected to use model DOFs. If `Case` DOFs are used, they are reexpanded to model DOFs using `def=struct('def',Case.T*def.def,'DOF`
  This is currently used to by the `*b.m` element family for geometrically non-linear matrices.
- `MatType` or `Opt` describing the desired output, appropriate handling of linear constraints, ect.

Output formats are

- `model` with the additional field `model.K` containing the matrices. The corresponding types are stored in `model.Opt(2,:)`. The `model.DOF` field is properly filled.
- `[m,k,mdof]` returning both mass and stiffness when `Opt(1)==0`
- `[Mat,mdof]` returning a matrix with type specified in `Opt(1)`, see `MatType` below.

`mdof` is the DOF definition vector describing the DOFs of output matrices.

When fixed boundary conditions or linear constraints are considered, `mdof` is equal to the set of master or independent degrees of freedom `Case.DOF` which can also be obtained with `fe_case(model,'gettdof')`. Additional unused DOFs can then

be eliminated unless `Opt(2)` is set to 1 to prevent that elimination. To prevent constraint elimination in `fe_mknl` use `Assemble NoT`.

In some cases, you may want to assemble the matrices but not go through the constraint elimination phase. This is done by setting `Opt(2)` to 2. `mdof` is then equal to `model.DOF`.

This is illustrated in the example below

```
femesh('reset');
model =femesh('testubeam');
model.DOF=[];% an non empty model.DOF would eliminate all other DOFs
model =fe_case(model,'fixdof','Base','z==0');
model = fe_mk(model,'Options',[0 2]);
[k,mdof] = fe_mk(model,'options',[0 0]);
fprintf('With constraints %i DOFs\n',size(k,1));
fprintf('Without          %i DOFs',size(model.K{1},1));
Case=fe_case(model,'gett');
isequal(Case.DOF,mdof) % mdof is the same as Case.DOF
```

For other information on constraint handling see section 5.13.

Assembly is decomposed in two phases. The initialization prepares everything that will stay constant during a non-linear run. The assembly call performs other operations.

Init

The `fe_mknl Init` phase initializes the `Case.T` (basis of vectors verifying linear constraints see section 5.13), `Case.GroupInfo` fields (detailed below) and `Case.MatGraph` (preallocated sparse matrix associated with the model topology for optimized (re)assembly). `Case.GroupInfo` is a cell array with rows giving information about each element group in the model (see section 5.14.1 for details).

`Case = fe_mknl('initNoCon', model)` can be used to initialize the case structure without building the matrix connectivity (sparse matrix with preallocation of all possible non zero values). `InitKeep` can be used to prevent changint the `model.DOF` DOF list. This is typically used for submodel assembly.

The initialization phase is decomposed into the following steps

- Generation of a complete list of DOFs using the `feutil('getdof',model)` call.
- get the material and element property tables in a robust manner. Generate node positions in a global reference frame.
- For each element group, build the `GroupInfo` data (DOF positions).
- For each element group, determine the unique pairs of `[MatId ProId]` values in the current group of elements and build a separate `integ` and `constit` for each pair. One then has the constitutive parameters for each type of element

in the current group. pointers rows 6 and 7 give for each element the location of relevent information in the integ and constit tables.

- For each element group, perform other initializations as defined by evaluating the callback string obtained using elem('GroupInit'). For example, intialize integration rule data structures, define local bases or normal maps, allocate memory for internal state variables...
- If requested (call without NoCon), preallocate a sparse matrix to store the assembled model. This topology assumes non zero values at all components of element matrices so that it is identical for all possible matrices and constant during non-linear iterations.

Assemble [, NoT]

The second phase, assembly, is optimized for speed and multiple runs (in non-linear sequences it is repeated as long as the element connectivity information does not change). In fe_mk the second phase is optimized for robustness. The following example illustrates the interest of multiple phase assembly

```
femesh('reset');
model =femesh('test hexa8 divide 100 10 10');
% traditional FE_MK assembly
tic;[m1,k1,mdof] = fe_mk(model);toc

% Multi-step approach for NL operation
tic;[Case,model.DOF]=fe_mknl('init',model);toc
tic;
m=fe_mknl('assemble',model,Case,2);
k=fe_mknl('assemble',model,Case,1);
toc
```

MatType

Matrix types are numeric indications of what needs to be computed during assembly. Currently defined types for OpenFEM are

- 0 mass and stiffness assembly. 1 stiffness, 2 mass, 3 viscous damping, 4 hysteretic damping, 5 tangent stiffness in geometric non-linear mechanics. Gyroscopic coupling 7 and stiffness 8 are supported in fe_cyclic. 9 is reserved for non-symmetric stiffness coupling (fluid structure, contact/friction, ...);
- 100 volume load, 101 pressure load, 102 inertia load, 103 initial stress load. Note that some load types are only supported with the mat_og element family;
- 200 stress at node, 201 stress at element center, 202 stress at gauss point
- 251 energy associated with matrix type 1 (stiffness), 252 energy associated with matrix type 2 (mass), ...

- `300` compute initial stress field associated with an initial deformation. This value is set in `Case.GroupInfo{jGroup,5}` directly (be careful with the fact that such direct modification INPUTS is not a MATLAB standard feature). `301` compute the stresses induced by a thermal field.

`Opt`

`fe_mk` options are given by calls of the form `fe_mk(model,'Options',Opt)` or the obsolete `fe_mk(node,elt,pl,il,[],adof,opt)`.

`opt(1)`	`MatType` see above
`opt(2)`	if active DOFs are specified using `model.DOF` (or the obsolete call with `adof`), DOFs in `model.DOF` but not used by the model (either linked to no element or with a zero on the matrix or both the mass and stiffness diagonals) are eliminated unless `opt(2)` is set to `1` (but case constraints are then still considered) or `2` (all constraints are ignored).
`opt(3)`	Assembly method (0 default, 1 symmetric mass and stiffness (OBSOLETE), 2 disk (to be preferred for large problems)). The disk assembly method creates temporary files using the MATLAB `tempname` function. This minimizes memory usage so that it should be preferred for very large models.
`opt(4)`	`0` (default) nothing done for less than 1000 DOF method 1 otherwise. `1` DOF numbering optimized using current `ofact SymRenumber` method. Since new solvers renumber at factorization time this option is no longer interesting.

Old formats `[m,k,mdof]=fe_mk(node,elt,pl,il)` returns mass and stiffness matrices when given nodes, elements, material properties, element properties rather than the corresponding model data structure.

`[mat,mdof]=fe_mk(node,elt,pl,il,[],adof,opt)` lets you specify DOFs to be retained with `adof` (same as defining a `Case` entry with `{'KeepDof', 'Retained', adof}`).

These formats are kept for backward compatibility but they do not allow support of local coordinate systems, handling of boundary conditions through cases, ...

Notes `fe_mk` no longer supports complex matrix assembly in order to allow a number of speed optimization steps. You are thus expected to assemble the real and imaginary parts successively.

See also Element functions in chapter 6, `fe_c`, `feplot`, `fe_eig`, `upcom`, `fe_mat`, `femesh`, etc.

fe_stres

Purpose Computation of stresses and energies for given deformations.

Syntax

```
Result = fe_stres('Command',MODEL,DEF)
  ...  = fe_stres('Command',node,elt,pl,il, ...)
  ...  = fe_stres( ... ,mode,mdof)
```

Description You can display stresses and energies directly using `fecom ColordataEnergies` and `ColordataEner` commands and use `fe_stres` to analyze results numerically. `MODEL` can be specified by four input arguments `node`, `elt`, `pl` and `il` (those used by `fe_mk`, see also section 5.1 and following), a structure array with fields `.Node`, `.Elt`, `.pl`, `.il`, or a database wrapper with those fields.

The deformations `DEF` can be specified using two arguments: `mode` and associated DOF definition vector `mdof` or a structure array with fields `.def` and `.DOF`.

`ene [m,k]`*`ElementSelection`*

Element energy computation. For a given shape, the levels of strain and kinetic energy in different elements give an indication of how much influence the modification of the element properties may have on the global system response. This knowledge is a useful analysis tool to determine regions that may need to be updated in a FE model.

The strain and kinetic energies of an element are defined by

$$E^e_{strain} = \frac{1}{2}\phi^T K_{element}\phi \quad \text{and} \quad E^e_{kinetic} = \frac{1}{2}\phi^T M_{element}\phi$$

Element energies for elements selected with *`ElementSelection`* (see the `femesh FindElt` commands) are computed for deformations in `DEF` and the result is returned in the structure array `RESULT` with fields `.data` and `.EltId` which specifies which elements were selected.

For comples frequency responses, one integrates the response over one cycle, which corresponds to summing the energies of the real and imaginary parts and using a factor 1/4 rather than 1/2.

`stress`

`out=fe_stres('stress `*`CritFcn Rest`*`',MODEL,DEF,`*`EltSel`*`)` returns the stresses evaluated at elements of `Model` selected by *`EltSel`*.

The *`CritFcn`* part of the command string is used to select a criterion. Currently supported criteria are

sI, sII, sIII	principal stresses from max to min. sI is the default.
mises	Returns the von Mises stress (note that the plane strain case is not currently handled consistently).
-com *i*	Returns the stress components of index *i*.

The Rest part of the command string is used to select a restitution method. Currently supported restitutions are

AtNode	average stress at each node (default). Note this is not currently weighted by element volume and thus quite approximate. Result is a structure with fields .DOF and .data
AtCenter	stress at center or mean stress at element stress restitution points. Result is a structure with fields .EltId and .data
AtInteg	stress at integration points (*b family of elements)
Gstate	returns a case with Case.GroupInfo{jGroup,5} containing the group gstate. This will be typically used to initialize stress states in for non-linear computations. For multiple deformations, gstate the first nElt columns correspond to the first deformation.

To obtain strain computations, use the strain material as shown below.

```
[model,def]=hexa8('testload stress');
model.pl=m_elastic('dbval 100 strain','dbval 112 strain');
model.il=p_solid('dbval 111 d3 -3');
data=fe_stres('stress atcenter',model,def)
```

See also fe_mk, feplot, fecom

fe_super

Purpose Generic element function for superelement support.

Description The non current *SDT 3* version of this function is included in OpenFEM. Use the `help fecom` command to get help.

See also `fesuper`, the `d_cms2` demonstration

iimouse

Purpose Mouse related callbacks for GUI figures.

Syntax

```
iimouse
iimouse('ModeName')
iimouse('ModeName',Handle)
```

Description The non current *SDT 3* version of this function is included in OpenFEM. Use the `help fecom` command to get help.

See also `iicom`, `fecom`, `iiplot`, `iiplot`

nopo

Purpose Imports nopo files (cf. Modulef)

Syntax

```
model      = nopo('read -v -p type FileName')
[Node,Elt] = nopo('read -v -p type FileName')
```

Description

`read`

The `-v` is used for verbose output. The optional -p type gives the type of problem described in the nopo file, this allows proper translation to OpenFEM element names. Supported types are

`'2D','3D','AXI','FOURIER','INCOMPRESSIBLE','PLAQUE','COQUE'`.

See also `medit`

medit

Purpose Export to Medit format

Syntax

```
[indnum,scale] = medit('write FileName',model)
[indnum,scale] = medit('write FileName',model,def,[opt])
[indnum,scale] = medit('write FileName',model,def,'a',[opt])
[indnum,scale] = medit('write FileName',model,[],strain)
[indnum,scale] = medit('write FileName',model,def,strain,[opt])
[indnum,scale] = medit('write FileName',model,def,strain,'a',[opt])
```

Description Medit is an interactive mesh visualization software, developed by the Gamma project at INRIA-Rocquencourt.

Medit executable is freely available at http://www-rocq.inria.fr/gamma/medit. Documentation can also be obtained.

medit is an interface to Medit software. This function creates files needed by Medit and runs the execution of these files in Medit. Users must download and install Medit themselves. It is not provided in OpenFEM.

Input arguments are the following :

FileName :

file name where information for Medit will be written, no extension must be given in FileName.

model :

a structure defining the model. It must contain at least fields .Node and .Elt.

def :

a structure defining deformations that users want to visualize. It must contain at least fields .def and .DOF.

strain :

structure defining coloring, must at least contain :

* fields .data and .EltId if coloring depends on elements

* fields .data and .DOF if coloring depends on nodes

Strain can be obtained by a call to fe_stres.
For example, strain=fe_stres('ener',FEnode,FEel0,pl,il,md1,mdof) generates a structure with fields .data and .EltId (depending on elements), and strain = fe_stres('stress mises',FEnode,FEel0,pl,[],def,mdof) generates a structure with fields .data and .DOF (depending on nodes). See demo d_ubeam or test test_medit.

`opt :`

option vector, `opt = [numdef nb_imag scale_user]` with

* `numdef` : mode to display number

* `nb_imag` : number of files to create the animation of deformations

* `scale_user` : display scale (a parameter for increasing the deformations)

`indnum :`

returns the nodes numbering used by Medit

`scale :`

returns the scale that was used to display the deformations

Use

`medit('write FileName',model) :`

displays the mesh defined by model

`medit('write FileName',model,def,[opt]) :`

displays the mesh defined by model in a window and the deformation defined by def in an other window

`medit('write FileName',model,def,'a',[opt]) :`

animates deformations defined by def on model

`medit('write FileName',model,[],strain) :`

displays the mesh defined by model and colors it with the help of strain

`medit('write FileName',model,def,strain,[opt]) :`

displays the mesh defined by model in a window and the deformation defined by def with colors due to strain in an other window

`medit('write FileName',model,def,strain,'a',[opt]) :`

animates the deformations defined by def on model and colors them with the help of strain

Note that nodes and faces references are given to Medit. You can visualize faces references by pressing the right mouse button, selecting "Shading" in the "Render mode" menu and then selecting "toggle matcolors" in the "Colors, Materials" menu.

of2vtk

Purpose Export model and deformations to VTK format for visualization purposes.

Syntax

```
opfem2VTK(FileName,model)
opfem2VTK(FileName,model,val1,...,valn)
```

Description Simple function to write the mesh corresponding to the structure model and associated data currently in the "Legacy VTK file format" for visualization.

To visualize the mesh using VTK files you may use ParaView which is freely available at http://www.paraview.org or any other visualization software supporting VTK file formats.

```
try;tname=nas2up('tempname.vtk');catch;tname=[tempname '.vtk'];end
model=femesh('testubeam');
of2vtk(tname,model);
```

The default extention .vtk is added if no extention is given;

Input arguments are the following:

FileName :

file name for the VTK output, no extension must be given in FileName, "FileName.vtk" is automatically created.

model :

a structure defining the model. It must contain at least fields .Node and .Elt. **FileName and model fields are mandatory**.

vali :

To create a VTK file defining the mesh and some data at nodes/elements (scalars, vectors) you want to visualize, you must specify as many inputs *vali* as needed. *vali* is a structure defining the data: $vali = struct('Name', ValueName, 'Data', Values)$. Values can be either a table of scalars ($Nnode \times 1$ or $Nelt \times 1$) or vectors ($Nnode \times 3$ or $Nelt \times 3$) at nodes/elements. Note that a deformed model can be visualized by providing nodal displacements as data (e.g. in ParaView using the "warp" function).

ofutil

Purpose OpenFEM utilities

Syntax

```
ofutil commands
```

Description This function is used for compilations, path checking, documentation generation, ...
Accepted commands are

Path	checks path consistency with possible removal of SDT
mexall	compiles all needed DLL
of_mk	compiles of_mk.f (see openfem/mex directory)
nopo2sd	compiles nopo2sd.c (located in openfem/mex directory)
sp_util	compiles sp_ufil.c
zip	creates a zip archive of the OpenFEM library
hevea	generates documentation with HEVEA
latex	generates documentation with LaTeX

ofact

Purpose

Factored matrix object.

Syntax

```
ofact
ofact('method MethodName');
kd=ofact(k); q = kd\b;  ofact('clear',kd);
kd=ofact(k,'MethodName')
```

Description

The factored matrix object `ofact` is designed to let users write code that is independent of the library used to solve static problems of the form $[K]\{q\} = \{F\}$. For FEM applications, choosing the appropriate library for that purpose is crucial. Depending on the case you may want to use full, skyline, or sparse solvers. Then whithin each library you may want to specify options (direct, iterative, in-core, out-of-core, parallel, ...).

Using the `ofact` object in your code, lets you specify method at run time rather than when writing the code. Typical steps are

```
ofact('method spfmex'); % choose method
kd = ofact(k);          % create object and factor
static = kd\b           % solve
ofact('clear',kd)       % clear factor when done
```

For single solves `static=ofact(k,b)` performs the three steps (factor, solve clear) in a single pass.

The first step of method selection provides an open architecture that lets users introduce new solvers with no need to rewrite functions that use `ofact` objects. Currently available methods are listed simply by typing

```
>> ofact

Available factorization methods for OFACT object
->  spfmex : SDT sparse LDLt solver
   sp_util : SDT skyline solver
        lu : MATLAB sparse LU solver
    mtaucs : TAUCS sparse solver
   pardiso : PARDISO sparse solver
      chol : MATLAB sparse Cholesky solver
   *psldlt : SGI sparse solver (NOT AVAILABLE ON THIS MACHINE)
```

and the method used can be selected with `ofact('method MethodName')`. SDtools maintains pointers to pre-compiled solvers at `http://www.sdtools.com/faq/FE_ofact.html`.

The factorization `kd = ofact(k);` and resolution steps `static = kd\b` can be separated to allow multiple solves with a single factor. Multiple solves are essential for eigenvalue and quasi-newton solvers. `static = ofact(k)\b` is of course also correct.

The clearing step is needed when the factors are not stored as MATLAB variables. They can be stored in another memory pile, in an out-of-core file, or on another computer/processor. Since for large problems, factors require a lot of memory. Clearing them is an important step.

Historically the object was called `skyline`. For backward compatibility reasons, a `skyline` function is provided.

umfpack

To use UMFPACK as an `ofact` solver you need to install it on your machine. This code is availlable at www.cise.ufl.edu/research/sparse/umfpack.

pardiso

For installation, see section 2.3.

Based on the Intel MKL (Math Kernel Library), you should use version 8 and after.

By default the pardiso call used in the ofact object is set for symmetric matrices. For non-symmetric matrices, you have to complement the ofact standard command for factorization with the character string `'nonsym'`. Moreover, when you pass a matrix from Matlab to PARDISO, you **must transpose** it in order to respect the PARDISO sparse matrix format.
Assuming that k is a real non-symmetric matrix and b a real vector, the solution q of the system $k.q = b$ is computed by the following sequence of commands:

```
ofact pardiso                    % select PARDISO solver
kd = ofact('fact nonsym',k');    % factorization
q=kd\b;                          % solve
ofact('clear',kd);               % clear ofact object
```

The factorization is composed of two steps: symbolic and numerical factorization. For the first step the solver needs only the sparse matrix structure (i.e. non-zeros location), whereas the actual data stored in the matrix are required in the second step only. Consequently, for a problem with a unique factorization, you can group the steps. This is done with the standard command *ofact('fact',...)*.
In case of multiple factorizations with a set of matrices having the same sparse structure, only the second step should be executed for each factorization, the first one is called just for the first factorization. This is possible using the commands `'symbfact'` and `'numfact'` instead of 'fact' as follows:

```
kd = ofact('symbfact',k);   % just one call at the beginning
...
kd = ofact('numfact',k,kd); % at each factorization
q=kd\b;                     %
...
ofact('clear',kd);          % just one call at the end
```

You can extend this to **non-symmetric systems** as described above.

Your solver To add your own solver simply add a file called `MySolver_utils.m` in the `@ofact` directory. This function must accept the commands detailed below.

Your object can use the fields `.ty` used to monitor what is stored in the object (0 unfactored ofact, 1 factored ofact, 2 LU, 3 Cholesky, 5 other), `.ind`, `.data` used to store the matrix or factor in true ofact format, `.dinv` inverse of diagonal (currently unused), `.l` L factor in `lu` decomposition or transpose of Cholesky factor, `.u` U factor in `lu` decomposition or Cholesky factor, `.method` other free format information used by the object method.

`method`

Is used to define defaults for what the solver does.

`fact`

This is the callback that is evaluated when `ofact` initializes a new matrix.

`solve`

This is the callback that is evaluated when `ofact` overloads the matrix left division (\)

`clear`

`clear` is used to provide a clean up method when factor information is not stored within the `ofact` object itself. For example, in persistent memory, in another process or on an another computer on the network.

See also `fe_eig`

sp_util

Purpose Sparse matrix utilities.

Description This function should be used as a `mex` file. The `.m` file version does not support all functionality, is significantly slower and requires more memory.

The `mex` code **is not** MATLAB **clean**, in the sense that it often modifies input arguments. You are thus not encouraged to call `sp_util` yourself.

The following comments are only provided, so that you can understand the purpose of various calls to `sp_util`.

`sp_util` with no argument returns its version number.

`sp_util('ismex')` true if `sp_util` is a `mex` file on your platform/path.

`ind=sp_util('profile',k)` returns the profile of a sparse matrix (assumed to be symmetric). This is useful to have a idea of the memory required to store a Cholesky factor of this matrix.

`ks=sp_util('sp2sky',sparse(k))` returns the structure array used by the `ofact` object.

`ks = sp_util('sky_dec',ks)` computes the LDL' factor of a ofact object and replaces the object data by the factor. The `sky_inv` command is used for forward/backward substitution (take a look at the `@ofact\mldivide.m` function). `sky_mul` provides matrix multiplication for unfactored ofact matrices.

`k = sp_util('nas2sp',K,RowStart,InColumn,opt)` is used by `nasread` for fast transformation between NASTRAN binary format and MATLAB sparse matrix storage.

`k = sp_util('spind',k,ind)` renumbering and/or block extraction of a matrix. The input and output arguments `k` MUST be the same. This is not typically acceptable behavior for MATLAB functions but the speed-up compared with `k=k(ind,ind)` can be significant.

`k = sp_util('xkx',x,k)` coordinate change for `x` a 3 by 3 matrix and DOFs of `k` stacked by groups of 3 for which the coordinate change must be applied.

`ener = sp_util('ener',ki,ke,length(Up.DOF),mind,T)` is used by `upcom` to compute energy distributions in a list of elements. Note that this function does not handle numerical round-off problems in the same way as previous calls.

`k = sp_util('mind',ki,ke,N,mind)` returns the square sparse matrix `k` associated to the vector of full matrix indices `ki` (column-wise position from `1` to `N^2`) and associated values `ke`. This is used for finite element model assembly by `fe_mk` and `upcom`. In the later case, the optional argument `mind` is used to multiply the blocks of `ke` by appropriate coefficients. `mindsym` has the same objective but assumes that `ki,ke` only store the upper half of a symmetric matrix.

`sparse = sp_util('sp2st',k)` returns a structure array with fields corresponding

to the MATLAB sparse matrix object. This is a debugging tool.

stack_get,stack_set,stack_rm

Purpose

Stack handling functions.

Syntax

```
[StackRows,index]=stack_get(model,typ);
[StackRows,index]=stack_get(model,typ,name);
Up=stack_set(model,typ,name,val)
Up=stack_rm(model,typ,name);
Up=stack_rm(model,typ);
Up=stack_rm(model,'',name);
```

Description

The `.Stack` field is used to store a variety of information, in a N by 3 cell array with each row of the form `{'type','name',val}` (see section 5.6 or section 5.7 for example). The purpose of this cell array is to deal with an unordered set of data entries which can be classified by type and name.

Since sorting can be done by name only, names should all be distinct although if the types are different this is not an obligation. In get and remove calls, `typ` and `name` can start by `#` to use a regular expression based matching (see `regexp` for details on regular expressions).

Syntax

```
Case.Stack={'DofSet','Point accel',[4.03;55.03];
            'DofLoad','Force',[2.03];
            'SensDof','Sensors',[4 55 30]'+.03};
% Replace first entry
Case=stack_set(Case,'DofSet','Point accel',[4.03;55.03;2.03]);
Case.Stack
% Add new entry
Case=stack_set(Case,'DofSet','P2',[4.03]);
Case.Stack
% Remove entry
Case=stack_rm(Case,'','Sensors');Case.Stack
% Get DofSet entries and access
[Val,ind]=stack_get(Case,'DofSet')
Case.Stack{ind(1),3} % same as Val{1,3}
% Regular expression match of entry
stack_get(Case,'','#P*')
```

Bibliography

[1] N. Lieven and D. Ewins, "A proposal for standard notation and terminology in modal analysis," *Int. J. Anal. and Exp. Modal Analysis*, vol. 7, no. 2, pp. 151–156, 1992.

[2] T. Hughes, *The Finite Element Method, Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall International, 1987.

[3] M. Géradin and D. Rixen, *Mechanical Vibrations. Theory and Application to Structural Dynamics.* John Wiley & Wiley and Sons, 1994, also in French, Masson, Paris, 1993.

[4] J. Imbert, *Analyse des Structures par Eléments Finis*. E.N.S.A.E. Cépaques Editions.

[5] J. Batoz, K. Bathe, and L. Ho, "A study of tree-node triangular plate bending elements," *Int. J. Num. Meth. in Eng.*, vol. 15, pp. 1771–1812, 1980.

Index