Structural Dynamics Toolbox FEMLink

For Use with MATLAB®

User's Guide
Version 5.2

Etienne Balmès
Jean-Michel Leclère

How to Contact SDTools

33 +1 41 13 13 57	Phone
33 +6 77 17 29 99	Fax
SDTools 44 rue Vergniaud 75013 Paris (France)	Mail

http://www.sdtools.com	Web
comp.soft-sys.matlab	Newsgroup
http://www.openfem.net	An Open-Source Finite Element Toolbox

support@sdtools.com	Technical support
suggest@sdtools.com	Product enhancement suggestions
info@sdtools.com	Sales, pricing, and general information

Structural Dynamics Toolbox User's Guide on May 27, 2005

Contents

Preface

1.1 Getting started

This section is intended for people who don't want to read the manual. It summarizes what you should know before going through the *SDT* demos to really get started.

The *SDT* demonstrations are located in the `sdtdemos` directory which for a proper installation should be in your MATLAB path. Executing `demosdt` at the MATLAB prompt will also add the demo directory to your path if needed. Many of these demonstrations are associated to manual pages. You can easily access the proper page with your favorite web browser by typing the `doc` commands listed in the demos at the MATLAB prompt.

The series of `gart..` demos cover a great part of the typical uses of the *SDT*. These demos are based on the test article used by the GARTEUR Structures & Materials Action Group 19 which organized a Round Robin exercise where 12 European laboratories tested a single structure between 1995 and 1997.

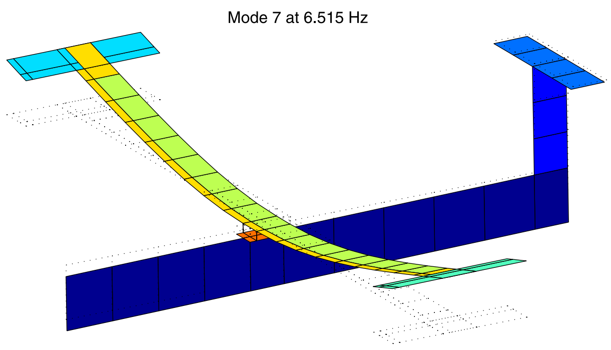


Figure 1.1: GARTEUR structure.

gartfe	builds the finite element model using the femesh preprocessor
gartte	shows how to prepare the visualization of test results and perform basic correlation
gartid	does the identification on a real data set
gartco	shows how to use fe_sens and fe_exp to perform modeshape expansion and more advanced correlation
gartsens	discusses sensor/shaker placement
gartup	shows how the upcom interface can be used to further correlate/update the model

The *SDT* provides tools covering the following areas.

Area 1: Experimental modal analysis

Experimental modal analysis combines techniques related to system identification (data acquisition and signal processing, done outside the *SDT*, followed parametric identification) with information about the spatial position of multiple sensors and actuators.

To get started with a modal analysis project read chapter 3.

Area 2: Test/analysis correlation

Correlation between test results and finite element predictions is a usual motivation for modal tests. Chapter 4 addresses topology correlation, test preparation, correlation criteria, modeshape expansion, and structural dynamic modification. Indications on how to use *SDT* for model updating are given in section 6.4.

Area 3: Basic finite element analysis

Finite element analysis capabilities of the *SDT* are now developed as part of the OpenFEM project. *SDT* extends this library with

- solvers for structural dynamics problems (eigenvalue, component mode synthesis, state-space model building, ...);
- solvers capable of handling large problems more efficiently than MATLAB;

- a complete set of tools for graphical pre/post-processing in an object oriented environment;
- high level handling of FEM solutions using cases;
- interface with other finite element codes through the FEMLink extension to *SDT*.

Chapter 5 gives a tutorial on FEM modeling in *SDT*. Developer information is given in chapter 7. Available elements are listed in chapter 8.

Area 4: Advanced FE analysis (model reduction, component mode synthesis, families of models)

Advanced model reduction methods are one of the key applications of *SDT*. To learn more about model reduction in structural dynamics read section 6.1. Typical applications are treated in section 6.2.

Finally, as shown in section 6.3, the *SDT* supports many tools necessary for finite element model updating.

1.2 Understanding the Toolbox architecture

1.2.1 Layers of code

The *SDT* has three layers of code.

- **Graphical user interfaces** (`feplot`, `iiplot`, `ii_mac`) provide a layer of pre-defined operations for Frequency Response Function (FRF) visualization and analysis, identification, 3-D deformation animation, and test/analysis correlation. Graphically supported operations (interactions between the user and plots/ menus/mouse movements/key pressed) form a subset of commands provided by user interface functions.

  The policy of the GUI layer is to let the user free to perform his own operations at any point. Significant efforts are made to ensure that this does not conflict with the continued use of GUI functions. But it is accepted that it may exceptionally do so, since command line and script access is a key to the flexibility of *SDT*. In most such cases, clearing the figure (using `clf`) or in the worst case closing it (use `close` or `delete`) and replotting will solve the problem.

- **User interface** (UI) functions provide high level solutions to problems in identification, finite element mesh handling, model reduction, sensor placement, superelement handling or parameterized models for FE model update. The first argument to these functions is a string command which is parsed to know what operations to perform. See `commode` for conventions linked to parsed commands.
- **Scientific functions** implement standard and state of the art methods in *experimental modal analysis*, *Finite Element analysis*, and to some extent in *structural design and FE model update*. These functions are open and can be easily extended to suit particular needs using the scientific environment provided by MATLAB.

1.2.2 Global variables

User interfaces require a knowledge of the current state of the interface and appropriate data. The policy of the *Toolbox* is to let the user free to perform his own operations at any point. Significant efforts are made to ensure that this does not conflict with the continued use of GUI functions, but it is accepted that it may exceptionally do so. This flexibility resulted in the use of both global variables (for information that the user is likely to modify) and graphical objects (for other information).

The user interface for data visualization and identification (`iicom`, `idcom`, `iiplot`) uses a number of standard global variables shown below

Frequency response data

`XF`	standard data base wrapper pointing to the global variables with the data and storing the characteristics of FRF data sets (see `xfopt` for details)
`IIw`	vector of frequency points (same as `XF(1).w`)
`IIxf`	MIMO set of measured FRF data (`XF(1).xf`)
`IIxe`	identified data set (`XF(2).xf`)
`IIxh`, `IIxi`	other sets of FRF data (`XF(3).xf` and `XF(4).xf`)
`IDopt`	identification options for the current model (see `idopt`, same as `XF(1).idopt`)
`XFdof`	global variable storing options describing each column of `XF(1).xf`

Identified model data

`IIpo`	set of poles of the main identified model (see `ii_pof`, same as `XF(5).po`)
`IIpo1`	set of poles for the alternate identified model (`XF(6).po`)
`IIres`	residues of the main identified model (see `idcom`, same as `XF(5).res`)
`IIres1`	residues of alternate identified model (see `idcom`, same as `XF(5).res`)

The `femesh` user interface for finite element mesh handling uses a number of standard global variables shown below

`FEnode`	main set of nodes (also used by `feplot`)
`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

By default, `femesh` and `iiplot` automatically use base workspace definitions of the standard global variables: base workspace variables with the correct name are transformed to `global` variables even if you did not dot it initially. 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`, `iiplot`, ... from your function which will declare the variables as global there too. The only thing that you should avoid is to use `clear` and not `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.

1.3 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

CMS	Component Mode Synthesis (see section 6.2.1)
COMAC	Coordinate Modal Assurance Criterion (see `ii_mac`)
DOF,DOFs	degree(s) of freedom (see section 7.5)
FE	finite element
MAC	Modal Assurance Criterion (see `ii_mac`)
MMIF	Multivariate Mode Indicator Function (see `ii_mmif`)
POC	Pseudo-orthogonality check (see `ii_mac`)

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 (see section 2.1). $\left\{\phi_j^T b\right\}, \left\{\psi_j^T b\right\}$ modal input matrix of the j^{th} normal / complex mode
$[c]$	sensor output shape matrix, model with N DOFs and NS outputs (see section 2.1). $\{c\phi_j\}, \{c\psi_j\}$ modal output matrix of the j^{th} normal / complex mode
$[E]_{NS\times NA}$	correction matrix for high frequency modes (see section 2.6)
$[F]_{NS\times NA}$	correction matrix for low frequency modes (see section 2.6)
M, C, K	mass, damping and stiffness matrices
N, NM	numbers of degrees of freedom, modes
NS, NA	numbers of sensors, actuators
$\{p\}_{NM\times 1}$	principal coordinate (degree of freedom of a normal mode model) (see section 2.2)
$\{q\}_{N\times 1}$	degree of freedom of a finite element model
s	Laplace variable ($s = i\omega$ for the Fourier transform)
$[R_j]$	$= \{c\psi_j\}\left\{\psi_j^T b\right\}$ residue matrix of the j^{th} complex mode (see section 2.6)
$[T_j]$	$= \{c\phi_j\}\left\{\phi_j^T b\right\}$ residue matrix of the j^{th} normal mode (used for proportionally damped models) (see section 2.6)
$\{u(s)\}_{NA\times 1}$	inputs (coefficients describing the time/frequency content of applied forces)
$\{y(s)\}_{NS\times 1}$	outputs (measurements, displacements, strains, stresses, etc.)
$[Z(s)]$	dynamic stiffness matrix (equal to $\left[Ms^2 + Cs + K\right]$)
$[\alpha(s)]$	dynamic compliance matrix (force to displacement transfer function)
p, α	design parameters of a FE model (see section 6.3.1)
$\Delta M, \Delta C, \Delta K$	additive modifications of the mass, damping and stiffness matrices (see section 6.3.1)
$[\Gamma]$	non-diagonal modal damping matrix (see section 2.3)
λ_j	complex pole (see section 2.5)
$[\phi]_{N\times NM}$	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 (see section 2.2)
$[\theta]_{N\times NM}$	NM complex modes of a first order symmetric structural model (see section 2.5)
$[\psi]_{N\times NM}$	NM complex modes of damped structural model (see section 2.5)

1.4 Release notes for SDT 5.2 and FEMLink 3.1

1.4.1 Key features

Key features of the SDT 5.2 release are

- MATLAB 7 compatibility. Fixes concern the disappearance of the `isglobal` function, the changes in the object inheritance properties, help integration.
- New support for the MATLAB 7.0 64 bit versions on Linux. This becomes the best platform to run large FEM models. The previous software limitation to 1.5 GB address space (4 GB on MACs) is no longer a difficulty.
- OpenFEM development has progressed with much improved handling of cases, a major revision of element functions to optimize model assembly and non-linear reassembly, bug fixes for load and stress computations.
- The new `sdthdf` functions implement a number of out-of-core operations that can be used when dealing with large FEM models.

Key features of FEMLink 3.1 are

- `nasread` optimization has continued resulting in major speed improvements for large FEM models and further robustness. Large `op2` file support has improved. `OUTPUT4` matrix reading has been compiled to allow for large matrix handling.
- `naswrite` has been rewritten in great part resulting in vastly improved speeds and major extensions in supported cards. The new `job` commands let the user drive NASTRAN from MATLAB. The new `EditBulk` commands can be used to generate multiple jobs. The new `wop4` command can be used to write matrices to `Output4` format.
- `ans2sdt` a number of bugs linked to the use of models combining symmetric and non symmetric element matrices have been fixed.
- `perm2sdt` the PERMAS reading interface has been fully rewritten allowing for much faster and consistent reading. Subcomponents are now supported.

For MATLAB compatibility see section 1.5.3.

1.4.2 Detail by function

`cbush`	the number of accepted input formats has been extended.
`fe_cyclic`	This new function supports cyclic symmetry : building of cyclic case entry, eigenvalue computation for n diameter modes.
`fe2ss`	Extensions on the types of sensors and loads supported. Bug fix on `upcom` assembly are returned basis.
`fe_sens`	Significant extensions of placement methodologies have been added see [2].
`flui4 ...`	this family of acoustic pressure element is now compiled.
`hexa8b ...`	this new family of elements (hexa20b, hexa27b, hexa8b, penta15b, penta6b, tetra10b, tetra4b) is the first series of the new generic multiphysic elements. It supports fully anisotropic elasticity for geometrically non linear problems. `fe_cyclic` supports gyroscopic and stress stiffening computations for models composed of elements in this family.
`ii_mac`	minor robustness enhancements, improved figure and colorbar generation, MATLAB 7 compatibility
`idopt`	Matlab 7 compatibility and minor fixes.
`fe_case`	New `Connection` commands allow the creation of complex kinematic connections.
`fe_eig`	Support of the `EigOpt` case information. Fixes on `FMAX` support in solution 5. Improved renumering strategies.
`fe_exp`	The fixed sensor modes are now returned as a deformation structure.
`fe_load`	The load assembly was fully revised to optimize the process for non linear operations.

`fe_mat`	The `convert` command for unit conversion has been significantly enhanced. Many internal changes have been introduced for `fe_mknl` assembly.
`fe_mk`	The assembly strategy has undergone a major revision with the introduction of `fe_mknl` for assembly in non-linear problems.
`fe_norm`	Memory usage has been optimized. Minor bug fixes are introduced.
`fe_reduc`	Now supports output of results as `struct` for use in superelements.
`fe_time`	The Newmark scheme as been optimized to support output resampling, intermediate saves, explicit computations, ...
`fesuper`	Minor corrections to `set` and `MakeComplete` commands.
`feplot`, `fecom`	Significant enhancement of the material and property visualization interfaces.
`sdthdf`	this new function allows
`upcom`	assembly is now performed in two steps so as to optimize out of core operation
`ufread, ufwrite`	speed and robustness improvements were introduced

1.4.3 Notes by MATLAB release

- MATLAB 6.5 and 7.0.x

  SDT 5.2 and `FEMLink 3.1` are developed for these versions of MATLAB and is fully compatible with them.

  MATLAB is no longer compiled as a 64 bit code on SGI. This has negative effects operation speed for large sparse matrices used in FEM problems. Some improvements were introduced in SDT 5.0 but best performance is obtained under MATLAB 5.3. This comment holds for MATLAB 6.0 too.

- MATLAB 6.1

  There are no known incompatibilities but tests are no longer systematically performed on this version of MATLAB.

  OpenGL support on LINUX has significant bugs so you may want to set the default `feplot` renderer to `zbuffer`

  ```
  cingui('Renderer zbuffer default')
  ```

- MATLAB 6.0, 5.x

  SDT is no longer tested and thus supported on these releases.

1.5 Release notes for SDT 5.1 and FEMLink 3.0

1.5.1 Key features

Key features of the SDT 5.1 release are

- The compilation of many time/memory intensive steps in FEM assembly, constraint elimination, static and eigenvalue solutions. As a result, much larger models and many more constraints can now be considered.
- Support for cyclic symmetry, non linear time integration, non-symmetric type 3 superelements (for fluid structure coupling in particular).
- OpenFEM development has progressed with much improved handling of cases, a major revision of element functions to optimize model assembly and non-linear reassembly, bug fixes for load and stress computations.
- A new wire-frame expansion method to interpolate sensor motion in unmeasured directions when non FEM model is available.

Key features of FEMLink 3.0 are

- `nasread`, `naswrite` full rewriting of NASTRAN interfaces with greatly enhanced reading speed, support for `DMIG` reading, optimization for large op2 files (up to 2GB), large numbers of material properties, support of more elements, improved handling of unknown cards, ...
- `naswrite` now supports `SPC`, `MPC`, `TABLED1`, `CORDi` information declared in a model. `DMIG`, element, material and property writing have been improved.
- `ans2sdt` support for non-symmetric ANSYS element matrices (found in fluid structure coupling problems) and corresponding support in type 3 superelements handled by SDT.
- `perm2sdt` support for a number of elements has been added to the PERMAS reading interface.

For MATLAB compatibility see section 1.5.3.

1.5.2 Detail by function

`celas`	All DOFs used by a `celas` are now retained by `feutil('getdof')` commands.
`femesh`, `feutil`	Now supports extrusion to `penta6` and a `facing` element selection. Minor improvements linked `object` and `test` commands. A number of changes linked to new `fe_mknl` assembly.
`fecom`, `feplot`	A number of minor enhancements and bug fixes : problems with arrows, cylindrical displacement coordinate systems, active axes. You can now enforce color limits using `cf.ua.clim=[min max];feplot`. The `iimouse cursor` context menu now supports a `3D-Line Pick` and `iiplot` display of response at current point.
`fe_ceig`	Now provide a real mode based estimate of complex modes with calls of the form `def=fe_ceig(model,eigopt)`.
`fe_case`	Major enhancements to constraint handling capabilities. New `GetData` and `GetTDof` commands. Support for cyclic symmetry. Constraint building is now handled by `fe_mpc` (which is called by the `fe_case GetT` command). Significant optimization efforts were made to handle models with thousands of rigid or multiple point constraints.
`fe_cyclic`	This new function supports cyclic symmetry : building of cyclic case entry, eigenvalue computation for n diameter modes.

`fe_load`	The load assembly was fully revised to optimize the process for non linear operations.
`fe_eig`	Speed and memory requirements have been optimized. Input form `{m,k,T,mdof}` and model data structures are now consistently accepted.
`fe_mat`	The `convert` command for unit conversion has been significantly enhanced. Many internal changes have been introduced for `fe_mknl` assembly.
`fe_mk`	The assembly strategy has undergone a major revision with the introduction of `fe_mknl` for assembly in non-linear problems.
`fe_reduc`	static and Craig-Bampton solvers were optimized to handle large problems.
`fe_sens`	A new `WireExp` command can be used to expand motion on all directions of the test wire frame.
`fesuper`	Underwent significant rewrite and optimization.
`fe_time`	Non linear problems are now supported : a Newmark scheme with a Newton loop is implemented.
`fsc`	Fluid-structure coupling element for a compressible, non-weighing fluid, with or without a free surface.
`idcom`	`ii_mac`, `iicom`, `matgui`, `propgui` a number of GUI related problems were resolved in these functions.
`nor2ss`	`nor2ss`, `nor2xf`, `nor2res` a number of minor bug fixes and enhancement to data structure output are introduced.
`ofact`	Skyline object has been changes in and supports : sparse and skyline matrices.
`qbode`	Problems with singular solutions at $s = 0$ are now treated properly.
`upcom`	now supports non-symmetric matrices. Energy densities can now be computed. A number of minor enhancement to parameter handling, file name defaults, ... were also introduced.

1.5.3 Notes by MATLAB release

- MATLAB 6.1 and 6.5

  SDT 5.1 and `FEMLink 3.0` are developed for these versions of MATLAB and is fully compatible with them.

  MATLAB is no longer compiled as a 64 bit code on SGI. This has negative effects operation speed for large sparse matrices used in FEM problems. Some improvements were introduced in SDT 5.0 but best performance is obtained under MATLAB 5.3. This comment holds for MATLAB 6.0 too.

- Matlab 6.0

  A bug in figure loading prevents reloading of `feplot` or `iiplot` figures.

  OpenGL support on LINUX has significant bugs so you may want to set the default `feplot` renderer to `zbuffer`

  ```
  cingui('Renderer zbuffer default')
  ```

- Matlab 5.x

  SDT is no longer tested and thus supported on these releases.

2

Structural dynamic concepts

This theoretical chapter is intended as a reference for the fundamental notions and associated variables used throughout the *SDT*. This information is grouped here and hypertext reference is given in the HTML version of the manual.

Models of dynamic systems are used for identification phases and links with control applications supported by other MATLAB toolboxes and SIMULINK. Key concepts and variables are

`b,c`	input/output shape matrices (`b,c,pb,cp` variables)
`nor`	normal mode models (`freq,damp,cp,pb` variables)
`damp`	damping for full and reduced models
`cpx`	complex mode models (`lambda, psi` variables)
`res`	pole/residue model (`res,po` variables)
`ss`	state space model (`a,b,c,d` variables)
`tf`	parametric transfer function (`num,den` variables)
`xf`	non-parametric transfer function (`w,xf` variables)

2.1 I/O shape matrices

Dynamic loads applied to a discretized mechanical model can be decomposed into a product $\{F\}_q = [b]\{u(t)\}$ where

- the **input shape matrix** $[b]$ is time invariant and characterizes spatial properties of the applied forces and
- the vector of inputs $\{u\}$ allows the description of the time/frequency properties.

Similarly it is assumed that the outputs $\{y\}$ (displacements but also strains, stresses, etc.) are linearly related to the model coordinates $\{q\}$ through the sensor **output shape matrix** ($\{y\} = [c]\{q\}$).

Input and output shape matrices are typically generated with `fe_c` or `fe_load`. Understanding what they represent and how they are transformed when model DOFs/states are changed is essential.

Linear mechanical models take the general forms

$$\begin{array}{c} \left[Ms^2 + Cs + K\right]_{N\times N}\{q(s)\} = [b]_{N\times NA}\{u(s)\}_{NA\times 1} \\ \{y(s)\}_{NS\times 1} = [c]_{NS\times N}\{q(s)\}_{N\times 1} \end{array} \tag{2.1}$$

in the frequency domain (with $Z(s) = Ms^2 + Cs + K$), and

$$\begin{array}{c} [M]\{\ddot{q}\} + [C]\{\dot{q}\} + [K]\{q\} = [b]\{u(t)\} \\ \{y(t)\} = [c]\{q(t)\} \end{array} \tag{2.2}$$

in the time domain.

In the model form (2.1), the first set of equations describes the evolution of $\{q\}$. The components of q are called Degrees Of Freedom (DOFs) by mechanical engineers and states in control theory. The second *observation* equation is rarely considered by mechanical engineers (hopefully the *SDT* may change this). The purpose of this distinction is to lead to the block diagram representation of the structural dynamics

$$\begin{tikzcd} {} \arrow[r, "\{u(s)\}"] & \boxed{[b]} \arrow[r, "\{F(s)\}"] & \boxed{\left[Ms^2+Cs+K\right]^{-1}} \arrow[r, "\{q(s)\}"] & \boxed{[c]} \arrow[r, "\{y(s)\}"] & {} \end{tikzcd}$$

which is very useful for applications in both control and mechanics.

In the simplest case of a point force input at a DOF q_l, the input shape matrix is equal to zero except for DOF l where it takes the value 1

$$[b_l] = \begin{bmatrix} \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \leftarrow l \tag{2.3}$$

Since $\{q_l\} = [b_l]^T\{q\}$, the transpose this Boolean input shape matrix is often called a *localization matrix*. Boolean input/output shape matrices are easily generated by `fe_c` (see the section on DOF selection page 151).

Input/output shape matrices become really useful when not Boolean. For applications considered in the *SDT* they are key to

- distributed FEM loads, see `fe_load`.
- test analysis correlation. Since you often have measurements that do not directly correspond to DOFs (accelerations in non global directions at positions that do not correspond to finite element nodes, see section 3.1.2).

- model reduction. To allow the changes to the DOFs q while retaining the physical meaning of the I/O relation between $\{u\}$ and $\{y\}$ (see section 6).

2.2 Normal mode models

The spectral decomposition is a key notion for the resolution of linear differential equations and the characterization of system dynamics. Predictions of the vibrations of structures are typically done for linear elastic structures or, for non-linear cases, refer to an underlying tangent elastic model.

Spectral decomposition applied to elastic structures leads to *modal analysis*. The main objective is to correctly represent low frequency dynamics by a low order model whose size is typically orders of magnitude smaller than that of the finite element model of an industrial structure.

The use of normal modes defined by the spectral decomposition of the elastic model and corrections (to account for the restricted frequency range of the model) is fundamental in modal analysis.

Associated models are used in the **normal mode model format**

$$\begin{gathered}\left[[I]\,s^2+[\Gamma]\,s+[\Omega^2]\right]\{p(s)\}=\left[\phi^T b\right]\{u(s)\}\\ \{y(s)\}=[c\phi]\,\{p(s)\}\end{gathered} \tag{2.4}$$

where the modal masses (see details below) are assumed to be unity.

The `nor2res`, `nor2ss`, and `nor2xf` functions are mostly based on this model form (see `nor2ss` theory section). They thus support a low level entry format with four arguments

`om`	*modal stiffness matrix* Ω^2. In place of a full modal stiffness matrix `om`, a vector of *modal frequencies* `freq` is generally used (**in rad/s** if `Hz` is not specified in the type string). It is then assumed that `om=diag(freq.^2)`. `om` can be complex for models with structural damping (see the section on damping page 30).
`ga`	*modal damping matrix* Γ (viscous). *damping ratios* `damp` corresponding to the modal frequencies `freq` are often used instead of the modal damping matrix `ga` (`damp` cannot be used with a full `om` matrix). If `damp` is a vector of the same size as `freq`, it is then assumed that `ga=diag(2*freq.*damp)`. If `damp` is a scalar, it is assumed that `ga=2*damp*diag(freq)`. The application of these models is discussed in the section on damping page 30).
`pb`	*modal input matrix* $\{\phi_j\}^T [b]$ (input shape matrix associated to the use of modal coordinates).
`cp`	*modal output matrix* $[c] \{\phi_j\}$ (output shape matrix associated to the use of modal coordinates).

Higher level calls, use a data structure with the following fields

`.freq`	frequencies (units given by `.fsc` field). This field may be empty if a non diagonal `nor.om` is defined.
`.om`	alternate definition for a non diagonal reduced stiffness. Nominally `om` contains `diag(freq.^2)`.
`.damp`	modal damping ratio. Can be a scalar or a vector giving the damping ratio for each frequency in `nor.freq`.
`.ga`	alternate definition for a non diagonal reduced viscous damping.
`.pb`	input shape matrix associated with the generalized coordinates in which `nor.om` and `nor.ga` are defined.
`.cp`	output shape matrix associated with the generalized coordinates in which `nor.om` and `nor.ga` are defined.
`.dof_in`	A six column matrix where each row describes a load by `[SensID NodeID nx ny nz Type]` giving a sensor identifier (integer or real), a node identifier (positive integer), the projection of the measurement direction on the global axes (if relevant), a `Type`.
`.lab_in`	A cell array of string labels associated with each input
`.dof_out`	A six column matrix describing outputs following the `.dof_in` format
`.lab_out`	A cell array of string labels associated with each output

General load and sensor definitions are then supported using cases (see section 5.2).

Transformations **to** other model formats are provided using `nor2ss` (state-space model), `nor2xf` (FRFs associated to the model in the `xf` format), and `nor2res` (complex residue model in the `res` format). The use of these functions is demonstrated in `demo_fe`.

Transformations **from** other model formats are provided by `fe2ss`, `fe_eig`, `fe_norm`, ... (from full order finite element model), `id_nor` and `res2nor` (from experimentally identified pole/residue model).

2.3 Damping

Models used to represent dissipation at the local material level and at the global system level should typically be different. Simple viscous behavior is very often not appropriate to describe material damping while a viscous model is appropriate in the normal mode model format (see details in Ref. [3]).

2.3.1 Viscous damping in the normal mode model form

In the normal mode form, viscous damping is represented by the modal damping matrix Γ which is typically used to represent all the dissipation effects at the system level.

Models with **modal damping** assume that a diagonal Γ is sufficient to represent dissipation at a system level. The non-zero terms of Γ are then usually expressed in terms of damping ratios $\Gamma_{jj} = 2\zeta_j\omega_j$. The damping ratio ζ_j are accepted by most *SDT* functions instead of a full Γ. The variable name `damp` is then used instead of `ga` in the documentation.

For a model with modal damping, the matrices in (6.6) are diagonal so that the contributions of the different normal modes are uncoupled and correspond exactly to the spectral decomposition of the model (see cpx page 35 for the definition of complex modes). The rational fraction expression of the dynamic compliance matrix (transfer from the inputs $\{u\}$ to displacement outputs $\{y\}$) takes the form

$$[\alpha(s)] = \sum_{j=1}^{N} \frac{\{c\phi_j\}\left\{b^T\phi_j\right\}^T}{s^2 + 2\zeta_j\omega_j s + \omega_j^2} = \sum_{j=1}^{N} \frac{[T_j]_{NS\times NA}}{s^2 + 2\zeta_j\omega_j s + \omega_j^2} \tag{2.5}$$

where the contribution of each mode is characterized by the pole frequency ω_j, damping ratio ζ_j, and the residue matrix T_j (which is equal to the product of the normal mode output shape matrix $\{c\phi_j\}$ by the normal mode input shape matrix $\left\{\phi_j^T b\right\}$).

Modal damping is used when lacking better information. One will thus often set a uniform damping ratio ($\zeta_j = 1\%$ or `damp = 0.01`) or experimentally determined damping ratios that are different for each pole (`po=ii_pof(po,3); damp=po(:,2);`).

Historically, modal damping was associated to the **proportional damping model** introduced by Lord Rayleigh which assumes the usefulness of a global viscously damped model with a dynamic stiffness of the form

$$[Z(s)] = \left[Ms^2 + (\alpha M + \beta K)s + K\right]$$

While this model indeed leads to a modally damped normal mode model, the α and β coefficients can only be adjusted to represent physical damping mechanisms over very narrow frequency bands.

Using a diagonal $[\Gamma]$ can introduce significant errors when normal mode coupling through the spatial distribution of damping mechanisms is possible. The condition

$$2\zeta_j\omega_j/|\omega_j - \omega_k| \ll 1$$

proposed by Hasselman [4], gives a good indication of when modal coupling will not occur. One will note that a structure with a group of modes separated by a few percent in frequency and levels of damping close to 1% does not verify this condition. The uncoupling assumption can however still be applied to blocks of modes [5].

A normal mode model with a full Γ matrix is said to be *non-proportionally damped* and is clearly more general/accurate than the simple modal damping model. The *SDT* leaves the choice between the non-proportional model using a matrix `ga` and the proportional model using damping ratio for each of the pole frequencies (in this case one has `ga=2*diag(damp.*freq)` or `ga=2*damp*diag(freq)` if a scalar uniform damping ratio is defined).

For identification phases, standard approximations linked to the assumption of modal damping are provided by (`id_rc`, `id_rm` and `res2nor`), while `id_nor` provides an original algorithm of the determination of a full Γ matrix. Theoretical aspects of this algorithm and details on the approximation of modal damping are discussed in [5]).

2.3.2 Damping in finite element models

Standard damped finite element models allow the incorporation of viscous and structural damping in the form of real C and complex K matrices respectively.

`fe_mk` could assemble a viscous damping matrix with user defined elements that would support matrix type 3 (viscous damping) using a call of the form `fe_mk(MODEL,'options',3)` (see section 7.14 for new element creation). But viscous damping models are rarely appropriate at the finite element level [3], so that it is not supported by any current SDT element.

Structural or hysteretic damping represents dissipation by giving a loss factor at the element level leading to a dynamic stiffness of the form

$$Z(s) = \left[Ms^2 + K + iB\right] = Ms^2 + \sum_{j=1}^{NE} \left[K_k^e\right] (1 + i\eta_k^e) \tag{2.6}$$

Such models are best handled using `upcom` (see section 6.3), rather than complex valued constitutive parameters which will not work with many element functions. The following example defines two loss factors for group 6 and other elements of the Garteur FEM model. Approximate damped poles are then estimated on the basis of real modes (better approximations are discussed in [6])

```
upcom('load GartUp'); upcom('plotelt'); cf=feplot;
upcom('ParStackreset');
upcom('ParStackadd k','Constrained Layer','group6');
upcom('ParStackadd k','Main structure','group~=6');
%      type cur min max vtype
par = [ 1   1.0 0.1 3.0   1
        1   1.0 0.1 3.0   1 ];
upcom('ParCoef',par);

% assemble using different loss factors for each parameter
B=upcom('assemble k coef .05 .01');
[m,k]=upcom('assemble coef 1.0 1.0');
Case=fe_case(Up,'getcase');

% Estimate damped poles on real mode basis
def=fe_eig({m,k,Case.DOF},[6 20 1e3]);
mr=def.def'*m*def.def;  % this is the identity
cr=zeros(size(mr));
kr=def.def'*k*def.def+i*(def.def'*B*def.def);
[psi,lambda]=fe_ceig(mr,cr,kr);
cf.def={def.def*psi,def.DOF,lambda/2/pi}
```

Note that in this model, the poles λ_j are not complex conjugate since the hysteretic damping model is only valid for positive frequencies (for negative frequencies one should change the sign of the imaginary part of K).

Given a set of complex modes you can compute frequency responses with res2xf, or simply use the modal damping ratio found with fe_ceig. Continuing the example, above one uses

```
Case=fe_case(Case,'Dofload','Point loads',[4.03;55.03], ...
             'SensDof','Sensors',[4 55 30]'+.03);
Sens=fe_case(Case,'sens'); Load=fe_load(Case);
np=size(mr,1);

RES=struct('res',[],'po',ii_pof(lambda(7:np)/2/pi,3), ...
    'idopt',idopt('new'));
RES.idopt.residual=2;RES.idopt.fitting='complex';
for j1=7:np % deal with flexible modes
 Rj=(Sens.cta*def.def*psi(:,j1)) * ... % c psi
     (psi(:,j1).'*def.def'*Load.def);  % psi^T b
 RES.res(j1-6,:)=Rj(:).';
end

% Rigid body mode residual
RES.res(end+1,:)=0;
for j1=1:6
 Rj=(Sens.cta*def.def(:,j1))*(def.def(:,j1)'*Load.def);
 RES.res(end,:)=RES.res(end,:)+Rj(:).';
end

iiplot;IIw=linspace(5,60,2048);
r1=res2xf(RES,IIw);IIxf=r1.xf;
IIxe=nor2xf(def,[zeros(6,1);RES.po(:,2)],Case,IIw,'hz');
damp=[zeros(6,1);RES.po(:,2)];
def.data=sqrt(real(def.data.^2)).*sqrt(1+i*damp*2);
IIxh=nor2xf(def,[],Case,IIw,'hz');
IIpo=RES.po;iicom(';iixfon;iixeon;iixhon;submagpha')
```

Note that the presence of rigid body modes, which can only be represented as residual terms in the pole/residue format (see section 2.6), makes the example more complex. The plot illustrates differences in responses obtained with true complex modes, viscous modal damping or hysteretic modal damping (case where one uses the pole of the true complex mode with a normal mode shape) . Viscous and hysteretic modal damping are nearly identical. With true complex modes, only channels 2 and 4 show a visible difference, and then only near anti-resonances.

To incorporate static corrections, you may want to compute complex modes on bases generated by `fe2ss`, rather than simple modal bases obtained with `fe_eig`.

The use of a constant loss factor can be a crude approximation for materials exhibiting significant damping. Methods used to treat frequency dependent materials are described in Ref. [7].

2.4 State space models

While normal mode models are appropriate for structures, **state-space models** allow the representation of more general linear dynamic systems and are commonly used in the *Control Toolbox* or SIMULINK. The standard form for state space-models is

$$\begin{aligned}\{\dot{x}\} &= [A]\,\{x(t)\} + [B]\,\{u(t)\}\\ \{y\} &= [C]\,\{x(t)\} + [D]\,\{u(t)\}\end{aligned} \tag{2.7}$$

with inputs $\{u\}$, states $\{x\}$ and outputs $\{y\}$. State-space models are represented in the *SDT*, as generally done in other Toolboxes for use with MATLAB, using four independent matrix variables `a`, `b`, `c`, and `d` (you should also take a look at the LTI state-space object of the *Control Toolbox*).

The natural state-space representation of normal mode models (2.4) is given by

$$\begin{gathered}\left\{\begin{matrix}\dot{p}\\ \ddot{p}\end{matrix}\right\} = \begin{bmatrix}0 & I\\ -\Omega^2 & -\Gamma\end{bmatrix}\left\{\begin{matrix}p\\ \dot{p}\end{matrix}\right\} + \begin{bmatrix}0\\ \phi^T b\end{bmatrix}\{u(t)\}\\ \{y(t)\} = [c\phi \quad 0]\left\{\begin{matrix}p\\ \dot{p}\end{matrix}\right\}\end{gathered} \tag{2.8}$$

Transformations to this form are provided by `nor2ss` and `fe2ss`. Another special form of state-space models is constructed by `res2ss`.

A state-space representation of the nominal structural model (2.1) is given by

$$\begin{gathered}\left\{\begin{matrix}\dot{q}\\ \ddot{q}\end{matrix}\right\} = \begin{bmatrix}0 & I\\ -M^{-1}K & -M^{-1}C\end{bmatrix}\left\{\begin{matrix}q\\ \dot{q}\end{matrix}\right\} + \begin{bmatrix}0\\ M^{-1}b\end{bmatrix}\{u(t)\}\\ \{y(t)\} = [c \quad 0]\left\{\begin{matrix}q\\ \dot{q}\end{matrix}\right\}\end{gathered} \tag{2.9}$$

The interest of this representation is mostly academic because it does not preserve symmetry (an useful feature of models of structures associated to the assumption of reciprocity) and because $M^{-1}K$ is usually a full matrix (so that the associated memory requirements for a realistic finite element model would be prohibitive). The *SDT* thus always starts by transforming a model to the normal mode form and the associated state-space model (2.8).

The transfer functions from inputs to outputs are described in the frequency domain by

$$\{y(s)\} = \left([C]\,[s\ I - A]^{-1}\,[B] + [D]\right)\{u(s)\} \tag{2.10}$$

assuming that $[A]$ is diagonalizable in the basis of **complex modes**, model (2.7) is equivalent to the diagonal model

$$\begin{array}{c} \left(s\,[I] - \left[\backslash \lambda_{j \backslash}\right]\right)\{\eta(s)\} = \left[\theta_L^T b\right]\{u\} \\ \{y\} = [c\theta_R]\,\{\eta(s)\} \end{array} \tag{2.11}$$

where the left and right modeshapes (columns of $[\theta_R]$ and $[\theta_L]$) are solution of

$$\{\theta_{jL}\}^T\,[A] = \lambda_j\,\{\theta_{jL}\}^T \quad \text{and} \quad [A]\,\{\theta_{jR}\} = \lambda_j\,\{\theta_{jR}\} \tag{2.12}$$

and verify the orthogonality conditions

$$[\theta_L]^T\,[\theta_R] = [I] \quad \text{and} \quad [\theta_L]^T\,[A]\,[\theta_R] = \left[\backslash \lambda_{j \backslash}\right] \tag{2.13}$$

The diagonal state space form corresponds to the partial fraction expansion

$$\{y(s)\} = \sum_{j=1}^{2N} \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s - \lambda_j} = \sum_{j=1}^{2N} \frac{[R_j]_{NS \times NA}}{s - \lambda_j} \tag{2.14}$$

where the contribution of each mode is characterized by the pole location λ_j and the residue matrix R_j (which is equal to the product of the complex modal output $\{c\theta_j\}$ by the modal input $\left\{\theta_j^T b\right\}$).

The partial fraction expansion (2.14) is heavily used for the identification routines implemented in the *SDT* (see the section on the pole/residue representation ref page 37.

2.5 Complex mode models

The standard spectral decomposition discussed for state-space models in the previous section can be applied directly to second order models of structural dynamics. The associated modes are called **complex modes** by opposition to **normal modes** which are associated to elastic models of structures and are always real valued.

Left and right eigenvectors, which are equal for reciprocal structural models, can be defined by the second order eigenvalue problem,

$$\left[M\lambda_j^2 + C\lambda_j + K\right]\{\psi_j\} = \{0\} \tag{2.15}$$

In practice however, mathematical libraries only provide first order eigenvalue solvers to that a transformation to the first order form is needed. Rather than the trivial state-space form (2.9), the following generalized state-space form is preferred

$$\begin{gathered}\begin{bmatrix} C & M \\ M & 0 \end{bmatrix}\begin{Bmatrix} \dot{q} \\ \ddot{q} \end{Bmatrix} + \begin{bmatrix} K & 0 \\ 0 & -M \end{bmatrix}\begin{Bmatrix} q \\ \dot{q} \end{Bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}\{u\} \\ \{y\} = \begin{bmatrix} c & 0 \end{bmatrix}\begin{Bmatrix} q \\ \dot{q} \end{Bmatrix}\end{gathered} \tag{2.16}$$

The matrices M, C and K being symmetric (assumption of reciprocity), the generalized state-space model (2.16) is symmetric. The associate left and right eigenvectors are thus equal and found by solving

$$\left(\begin{bmatrix} C & M \\ M & 0 \end{bmatrix}\lambda_j + \begin{bmatrix} K & 0 \\ 0 & -M \end{bmatrix}\right)\{\theta_j\} = \{0\} \tag{2.17}$$

Because of the specific block from of the problem, it can be shown that

$$\{\theta_j\} = \begin{Bmatrix} \psi_j \\ \psi_j\lambda_j \end{Bmatrix} \tag{2.18}$$

where it should be noted that the name complex modeshape is given to both θ_j (for applications in system dynamics) and ψ_j (for applications in structural dynamics).

The initial model being real, complex eigenvalues λ_j come in conjugate pairs associated to conjugate pairs of modeshapes $\{\psi_j\}$. With the exception of systems with real poles, there are $2N$ complex eigenvalues for the considered symmetric systems ($\psi_{[N+1...2N]} = \bar{\psi}_{[1...N]}$ and $\lambda_{[N+1...2N]} = \bar{\lambda}_{[1...N]}$).

The existence of a set of $2N$ eigenvectors is equivalent to the verification of two orthogonality conditions

$$\begin{aligned} [\theta]^T\begin{bmatrix} C & M \\ M & 0 \end{bmatrix}[\theta] &= \psi^T C\psi + \Lambda\psi^T M\psi + \psi^T M\psi\Lambda &&= \left[\setminus I_\setminus\right]_{2N} \\ [\theta]^T\begin{bmatrix} K & 0 \\ 0 & -M \end{bmatrix}[\theta] &= \psi^T K\psi - \Lambda\psi^T M\psi\Lambda &&= -\left[\setminus \Lambda_\setminus\right]_{2N} \end{aligned} \tag{2.19}$$

where in (2.19) the arbitrary diagonal matrix was chosen to be the identity because it leads to a normalization of complex modes that is equivalent to the collocation constraint used to scale experimentally determined modeshapes ([5] and section 3.4.2).

Note that with hysteretic damping (complex valued stiffness, see section 2.3.2) the modes are not complex conjugate but opposite. To use a complex mode basis one thus needs to replace complex modes whose poles have negative imaginary parts with the conjugate of the corresponding mode whose pole has a positive imaginary part.

For a particular dynamic system, one will only be interested in predicting or measuring how complex modes are excited (modal input shape matrix $\left\{\theta_j^T B\right\} = \left\{\psi_j^T b\right\}$) or observed (modal output shape matrix $\{C\theta_j\} = \{c\psi_j\}$).

In the structural dynamics community, the **modal input shape matrix** is often called **modal participation factor** (and noted L_j) and the modal output shape matrix simply **modeshape**. A different terminology is preferred here to convey the fact that both notions are dual and that $\left\{\psi_j^T b_l\right\} = \{c_l \psi_j\}$ for a reciprocal structure and a collocated pair of inputs and outputs (such that $u\dot{y}$ is the power input to the structure).

For predictions, complex modes can be computed from finite element models using `fe_ceig`. Computing complex modes of full order models is not necessary. You should thus reduce the model on a basis of real vectors (as discussed in [6] and illustrated in section 2.3.2) before calling `fe_ceig` (if you really want complex modes).

For identification phases, complex modes are used in the form of residue matrices product $[R_j] = \{c\psi_j\}\left\{\psi_j^T b\right\}$ (see the next section). Modal residues are obtained by `id_rc` and separation of the modal input and output parts is obtained using `id_rm`.

For lightly damped structures, imposing the modal damping assumption, which forces the use of real modeshapes, may give correct result and simplify your identification work very much. Refer to section 3.4.3 for more details.

2.6 Pole/residue models

The spectral decomposition associated to complex modes, leads to a representation of the transfer function as a sum of modal contributions

$$[\alpha(s)] \;=\; \sum_{j=1}^{2N} \left(\frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j} \right) = \sum_{j=1}^{2N} \left(\frac{[R_j]}{s-\lambda_j} \right) \tag{2.20}$$

For applications in identification from experimental data, one can only determine modes whose poles are located in the test frequency range. The full series thus need to be truncated. The contributions of out-of-band modes cannot be neglected for applications in structural dynamics. One thus introduces a high frequency residual correction for truncated high frequency terms and, when needed, (quite often for suspended test articles) a low frequency residual for modes below the measurement frequency band.

These corrections depend on the type of transfer function so that the *SDT* uses the `IDopt` variable (see the reference section on the `idopt` function) to define the current type. `IDopt.Residual` specifies which corrections are needed (the default is 3 which includes both a low and high frequency residuals). `IDopt.Data` specifies if the FRF is force to displacement, velocity or acceleration. For a force to displacement transfer function with low and high frequency correction), the **pole/residue model** (also called partial fraction expansion) thus takes the form

$$[\alpha(s)] \;=\; \sum_{j\in\texttt{identified}} \left(\frac{[R_j]}{s-\lambda_j} + \frac{[\bar{R}_j]}{s-\bar{\lambda}_j} \right) + [E] + \frac{[F]}{s^2} \tag{2.21}$$

The *SDT* always stores pole/residue models in the displacement/force format. The expression of the force to acceleration transfer function is thus

$$[A(s)] \;=\; \sum_{j\in\texttt{identified}} \left(\frac{s^2\,[R_j]}{s-\lambda_j} + \frac{s^2\,[\bar{R}_j]}{s-\bar{\lambda}_j} \right) + s^2\,[E] + [F] \tag{2.22}$$

The **nominal** pole/residue model above is used when `IDopt.Fit='Complex'`. This model assumes that complex poles come in conjugate pairs and that the residue matrices are also conjugate which is true for real system.

The **complex residues with asymmetric pole structure** (`IDopt.Fit='Posit'`) only keep the poles with positive imaginary parts

$$[\alpha(s)] \;=\; \sum_{j\in\texttt{identified}} \left(\frac{[R_j]}{s-\lambda_j} \right) + [E] + \frac{[F]}{s^2} \tag{2.23}$$

which allows slightly faster computations when using `id_rc` for the identification but not so much so that the symmetric pole pattern should not be used in general. This option is only maintained for backward compatibility reasons.

The **normal mode residues with symmetric pole structure** (`IDopt.Fit='Nor'`)

$$[\alpha(s)] \;=\; \sum_{j\in\texttt{identified}} \left(\frac{[T_j]}{s^2+2\zeta_j\omega_j s+\omega_j^2} \right) + [E] + \frac{[F]}{s^2} \tag{2.24}$$

can be used to identify normal modes directly under the assumption of modal damping (see damp page 30).

Further characterization of the properties of a given pole/residue model is given by a structure detailed under the `xfopt Shapes at DOF` section.

The residue matrices `res` are stored using one row for each pole or asymptotic correction term and, as for FRFs (see the `xf` format), a column for each SISO transfer function (stacking NS columns for actuator 1, then NS columns for actuator 2, etc.).

$$\texttt{res} = \begin{bmatrix} \vdots & & \dots & \dots & & \dots \\ R_{j(11)} & R_{j(21)} & \dots & R_{j(12)} & R_{j(22)} & \dots \\ \vdots & & \ddots & \vdots & & \ddots \\ E_{11} & E_{21} & \dots & E_{12} & E_{22} & \dots \\ F_{11} & F_{21} & \dots & F_{12} & F_{22} & \dots \end{bmatrix} \tag{2.25}$$

The normal mode residues (`IDopt.Fit='Normal'`) are stored in a similar fashion with for only difference that the T_j are real while the R_j are complex.

2.7 Parametric transfer function

Except for the `id_poly` and `qbode` functions, the *SDT* does not typically use the numerous variants of the ARMAX model that are traditional in system identification applications and lead to the ratio of polynomials called transfer function format (`tf`) in other MATLAB *Toolboxes*. In modal analysis, transfer functions refer to the functions characterizing the relation between inputs and outputs. The `tf` format thus corresponds to the parametric representations of sets of transfer functions in the form of a ratio of polynomials

$$H_j(s) = \frac{a_{j,1}s^{na-1} + a_{j,2}s^{na-2} + \ldots + a_{j,na}}{b_{j,1}s^{nb-1} + b_{j,2}s^{nb-2} + \ldots + b_{j,nb}} \tag{2.26}$$

The *SDT* stacks the different numerator and denominator polynomials as rows of numerator and denominator matrices

$$\texttt{num} = \begin{bmatrix} a_{11} & a_{12} & \dots \\ a_{21} & a_{22} & \dots \\ \vdots & & \ddots \end{bmatrix} \text{ and } \texttt{den} = \begin{bmatrix} b_{11} & b_{12} & \dots \\ b_{21} & b_{22} & \dots \\ \vdots & & \ddots \end{bmatrix} \tag{2.27}$$

Other MATLAB toolboxes typically only accept a single common denominator (`den` is a single row). This form is also accepted by `qbode` which is used to predict FRFs at a number of frequencies in the non-parametric `xf` format).

The `id_poly` function identifies polynomial representations of sets of test functions and `res2tf` provides a transformation between the pole/residue and polynomial representations of transfer functions.

2.8 Non-parametric transfer function

For a linear system at a given frequency ω, the response vector $\{y\}$ at NS sensor locations to a vector $\{u\}$ of NA inputs is described by the NS by NA rectangular matrix of Frequency Responses (FRF)

$$\left\{ \begin{array}{c} y_1(\omega) \\ \vdots \\ y_{NS}(\omega) \end{array} \right\} = [H]\,\{u\} = \left[\begin{array}{ccc} H_{11}(\omega) & H_{12}(\omega) & \ldots \\ H_{21}(\omega) & H_{22}(\omega) & \\ \vdots & & \ddots \end{array} \right]_{NS\times NA} \left\{ \begin{array}{c} u_1(\omega) \\ \vdots \\ u_{NA}(\omega) \end{array} \right\} \tag{2.28}$$

The *SDT* stores frequencies at which the FRF are evaluated as a column vector `w`

$$\mathtt{w} = \left\{ \begin{array}{c} \omega_1 \\ \vdots \\ \omega_{NW} \end{array} \right\}_{NW\times 1} \tag{2.29}$$

and SISO FRFs H_{ij} are stored as columns of the matrix `xf` where each row corresponds to a different frequency (indicated in `w`). By default, it is assumed that the correspondence between the columns of `xf` and the sensors and actuator numbers is as follows. The NS transfer functions from actuator 1 to the NS sensors are stored as the first NS columns of `xf`, then the NS transfer functions of actuator 2, etc.

$$\mathtt{xf} = \left[\begin{array}{cccccc} H_{11}(\omega_1) & H_{21}(\omega_1) & \ldots & H_{12}(\omega_1) & H_{22}(\omega_1) & \ldots \\ H_{11}(\omega_2) & H_{21}(\omega_2) & \ldots & H_{12}(\omega_2) & H_{22}(\omega_2) & \ldots \\ \vdots & & \ddots & \vdots & & \ddots \end{array} \right]_{NW\times(NS\times NA)} \tag{2.30}$$

Further characterization of the properties of a given set of FRFs is given by a structure detailed under the `xfopt Response data` section.

Frequency response functions corresponding to parametric models can be generated in the `xf` format using `qbode` (transformation from `ss` and `tf` formats), `nor2xf`, or `res2xf`. These functions use robustness/speed trade-offs that are different from algorithms implemented in other MATLAB toolboxes and are more appropriate for applications in structural dynamics.

Modal test tutorial

An experimental modal analysis project can be decomposed in following steps

- before the test, preparation and design (see section 3.1)
- acquisition of test data, import into the SDT, direct exploitation of measurements (visualization, operational deflexion shapes, ...) (see section 3.2)
- identification of modal properties from test data (see section 3.3)
- handling of MIMO tests and other model transformations (output of identified models to state-space, normal mode, ... formats, taking reciprocity into account, ...) (see section 3.4)

Further steps (test/analysis correlation, shape expansion, structural dynamics modification) are discussed in chapter section 4.

3.1 Preparing a modal test

Before actually taking measurements, it is good practice to prepare a wire frame-display (section 3.1.1) and define the sensor configuration (section 3.1.2). The information takes the form of a test specific `.m` file which should look like the `gartte` demo without the various plot commands. The `d_pre` demo also talks about test preparation.

3.1.1 Geometry declaration

A wire-frame model is composed of node and connectivity declarations. The associated script ends by plotting commands. Based on the authors' experience, the easiest method to create a test geometry is to have a script with the elements shown below. This however requires familiarity with the process so that alternatives are gradually introduced and accessible from the `feplot` `File:New model ...` menu.

On Windows, the *SDT* you can currently edit nodes and connectivity using Excel (select that item in the `File:New model ...` list). Excel icons displayed in the `feplot` figure to let you update the plot based on the current worksheet content and close the Excel server when done. Note that this editor is initialized with the model currently showing in the `feplot` figure.

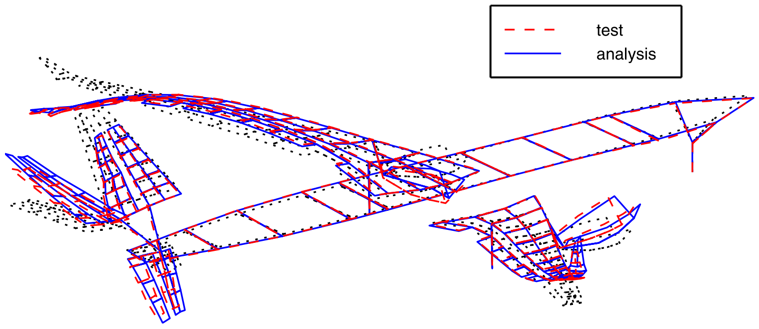


Figure 3.1: Test analysis : wire-frame model.

The node matrix specifies the 3-D location of the needed nodes following the standard node format (see section 7.1). For example, the node matrix linked to the 2-bay truss demonstration (see section 5.1.2) is

```
test.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];
```

The **connectivity** is the line used to connect the various test nodes. You could use any FE model for a connectivity but in general wire-frame representations are relatively sparse so that the *SDT* supports a special declaration format (the line connectivity matrix ldraw which corresponds to the Universal Format 82 (Trace Line)). Each line is a series of connected node numbers with 0 used to have discontinuous segments. It is good practice to use one line per substructure (so that you remove certain substructures from the display using the fecom group commands). For example, the 2-bay truss (see section 5.1.2) can be represented as two cells using

```
L=[1 3 2 4 3];
ldraw(1,[1 82+[1:length(L)]]) = [length(L) L];
st='Group1'; ldraw(1,3:length(st)+2)=st;
L=[3 6 0 6 5 0 4 5 0 4 6];
ldraw(2,[1 82+[1:length(L)]]) = [length(L) L];
st='Group2'; ldraw(2,3:length(st)+2)=st;
```

```
% transforms the trace into beam elements
test.Elt=feutil('trace2elt',ldraw);
```

Declaring the nodes and connectivity matrix is **a time consuming part** of the preparation of a modal test.

Once the `node`, `elt` or `ldraw` defined, you can visualize your test mesh using

```
cf=feplot;cf.model=test;
```

which shows the structure in its undeformed configuration. Note that

- you can start with an empty `ldraw` so that `feplot` draws a cloud of nodes. You can then use `fecom TextNode` or the mouse cursor (see `iimouse`) to see node numbers a gradually connect them by filling the `.m` script associated to your test.

  The `fe_fmesh('3dlineinit')` command will also start an automated line meshing tool cursor in the current `feplot` figure. Click on nodes continue the line, while the context menu allows breaks, last point removal, exit, and display of the `ldraw` building commands in the MATLAB command window. This procedure is particularly useful if you already have a FEM model of your test article.
- you can also use the `femesh ObjectBeamLine` command to define the experimental mesh.
- If you have a FE mesh, you can easily combine the FE and wire frame models using the `femesh AddTest` command (see `gartte`). The procedure to solve the common problem of matching coordinate systems when they initially have different scales / orientations / origins is discussed in section 4.1.1.

The `feplot` and `fecom` functions provide a number of tools that are designed to help in visualizing test results. You should take the time to go through the `gartid`, `gartte` and `gartco` demos to learn more about them.

3.1.2 Sensor/shaker configurations

The *SDT* handles translation and rotation sensors at arbitrary locations. The underlying principles detailed in section 4.1.2 are also applicable to strain sensors but this capability is not fully supported.

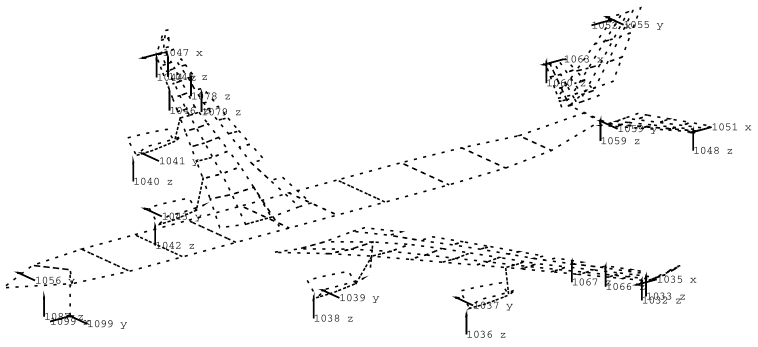


Figure 3.2: Sensor/shaker locations.

Basic sensor configurations correspond to cases where the measurements are translations in global directions and, if test analysis correlation is desired, the sensors are located at finite element nodes. In such cases, everything can be easily handled using DOF definition vectors.

DOF definition vectors (see `mdof` page 146 allow the description of translation DOFs in global directions. The convention that DOFs `.07` to `.09` correspond to translations in the $-x, -y, -z$ directions is implemented specifically for the common case where test sensors are oriented this way. For example, you can display sensors using (see the `gartte` demo)

```
cf=demosdt('demogartteplot')
sdof = [1011.03 1001.03 2012.07 1012.03 2005.07 1005.03 1008.03  ...
 1111.03 1101.03 2112.07 1112.03 2105.07 1105.03 1108.03 1201.07 ...
 2201.08 3201.03 1206.03 1205.08 1302.08 2301.07 1301.03 2303.07 ...
 1303.03]';
cf.sens(1) = sdof;  cf.o(3)='ty7sel1';
fecom(';scd.15;textdof');
```

`fe_sens` and `feplot` handle translation sensors in non-global directions by defining a 5 column matrix with rows containing `[SensID NodeID nx ny nz]` giving a sensor identifier (integer or real), a node identifier (positive integer), and the projection of the measurement direction on the global axes. This procedure does not require the cumbersome definition of multiple local coordinate systems. In the `gartte` example, the format would be

```
cf=demosdt('demogartteplot')
%        Address Node nx ny nz
```

```
sdof=[ 1011.03  1011   0.0   0.0   1.0;
        1001.03  1001   0.0   0.0   1.0;
        2012.07  2012  -1.0   0.0   0.0;
        1012.03  1012   0.0   0.0   1.0;
        2005.07  2005  -1.0   0.0   0.0;
        1005.03  1005   0.0   0.0   1.0;
        1008.03  1008   0.0   0.0   1.0;
        1111.03  1111   0.0   0.0   1.0];
cf.sens(1) = sdof;  cf.o(3)='ty7sel1';
fecom(';scd.15;textdof');
```

Sensor and shaker definitions associated with each measured input/output pair are normally entered in the acquisition phase as detailed in section 3.2.2. Except for roving hammer tests, the number of input locations is usually small and only used for MIMO identification (see section 3.4).

Once a sensor configuration defined, you can directly animate measured shapes (called Operational Deflection Shapes) as detailed in section 3.2.4.

3.2 Data import and visualization

3.2.1 Data acquisition

The *Structural Dynamics Toolbox* does not intend to support the acquisition of test data since tight integration of acquisition hardware and software is mandatory. The following table gives a partial list of systems with which the *SDT* has been successfully interfaced.

Vendor	Procedure used
Bruel & Kjaer	Export data from Pulse to the UFF and read into *SDT* with ufread or use the Bridge To Matlab software and pulse2sdt.
Dactron	Export data from RT-Pro software to the UFF. Use the *Active-X API* to drive the Photon from MATLAB.
LMS	Export data from LMS CADA-X to UFF.
MathWorks	Use *Data Acquisition* and *Signal Processing* toolboxes to estimate FRFs and create a script to fill in *SDT* information (see section 3.2.2).
MTS	Export data from IDEAS-Pro software to UFF.
Polytec	Export data from PSV software to UFF.
Spectral Dynamics	Create a Matlab script to format data from SigLab to *SDT* format.

You can find theoretical information on data acquisition for modal analysis in Refs. [8][9][10][11].

3.2.2 Importing FRF data

There are two main mechanisms to import FRF data into SDT. Universal files are easiest if generated by the acquisition system. Writting of an import script defining fields used by SDT is also fairly simple and described below (you can then use ufwrite to generate universal files for export).

The ufread and ufwrite functions allow conversions between the xf format and files in the Universal File Format which is supported by most measurement systems. A typical call would be

```
UFS=ufread('FileName.unv');  % read
iiplot                       % initialize iiplot
XF(1)=UFS(3)                 % show UFS(3) in iiplot
```

where you read the database wrapper UFS (see xfopt), initialize the standard database wrapper XF used by iiplot and idcom, assign dataset 3 of UFS to dataset 1 of XF (assuming that dataset three represents frequency response functions of interest).

Note that some acquisition systems write many universal files for a set of measurements (one file per channel). This is now supported by ufread with a stared file name

```
UFS=ufread('FileRoot*.unv');
```

Measured frequency responses are stored in the `.xf` field (frequencies in `.w`) and should comply with the specifications of the `xf` format (see details under `xf` page 40). Other fields needed to specify the physical meaning of each FRF are detailed in the `xfopt` reference section. When importing data from your own format or using a universal file where some fields are not correct, the *SDT* will generally function with default values set by the `xfopt` function, but you should still complete/correct these variables as much as possible.

For correct display of deformations and title/legend generation, you should set the `XF(1).dof` field (see more details in the `xfopt` response data section). For example one can consider a MIMO test with 2 inputs and 20 outputs stored as columns of variable `xf` with the rows corresponding to frequencies stored in `w`. You script will look like

```
XF(1).w=w;
XF(1).xf=xf;  % define the responses at all DOFs
out_dof=[1:20]+.03'; % output dofs for 20 sensors in y direction
in_dof=[1.03 10.03]; % input dofs for two shakers at nodes 1 and 10
out_dof=out_dof(:)*ones(1,length(in_dof));
in_dof=ones(length(out_dof),1)*in_dof(:)';
XF(1).dof(:,1:2)=[out_dof(:) in_dof(:)];
XF(1).idopt.nsna=size(out_dof);
XF(1).idopt.recip='mimo';
```

You can also edit these values using the `iiplot properties:channel` tab.

For correct identification using `id_rc`, you should verify the fields of `XF(1).idopt`. These correspond to the `IDcomGUI:Options` tab (see section 3.3). You can also edit these values in a script. For correct identification, you should set

```
demosdt('demogartid');
XF(1).idopt.Residual='3';
XF(1).idopt.DataType='Acc';
XF(1).idopt.Absci='Hz';XF(1).idopt.PoleU='Hz';
iicom('wmin 6 40') % sets XF(1).idopt.Selected
XF(1).idopt.Fit='Complex';
XF(1).idopt  % display current options
```

For correct transformations using `id_rm`, you should also verify `IDopt.NSNA` (number of sensors/actuators), `IDopt.Reciprocity` and `IDopt.Collocated`.

For correct labels using `iiplot` you should set the abscissa, and ordinate numerator/denominator types in the data base wrapper. You can edit these values using the `iiplot properties:channel` tab. A typical script would declare frequencies, acceleration, and force using (see list with `xfopt _datatype`)

```
UFS(2).x='Freq';UFS(2).yn='Acc';UFS(2).yd='Load';UFS(2).info
```

3.2.3 Getting started with the iiplot interface

Most identification problems should be solved using the standard commands for identification provided in idcom while running the iiplot interface for data visualization. To perform an identification correctly, you need to have some familiarity with the interface and in particular with the iicom commands that let you modify what you display. You should also take a look at iimouse which is used to enable mouse and key press operations within iiplot, feplot, and ii_mac figures.

To familiarize yourself with the iiplot interface, run demosdt('demogartidpro'). Which opens the iiplot figure and the associated iiplot(2) properties figure.

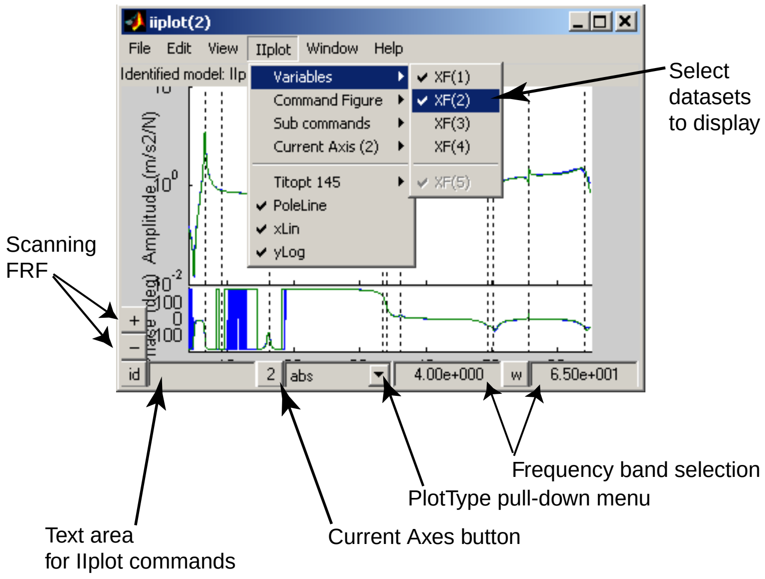
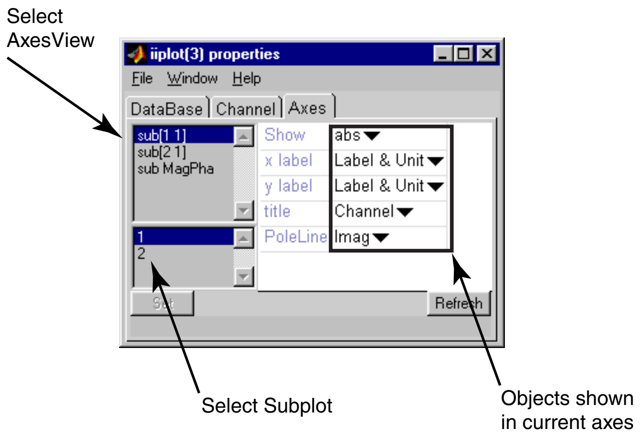


Figure 3.3: iiplot interface.



Show: abs
x label: Label & Unit
y label: Label & Unit
title: Channel
PoleLine: Imag

Figure 3.4: iiplot properties tab.

`iiplot` displays data stored in the global variable `XF` which contains four response datasets (`XF(1)` to `XF(4)`) and two shape datasets (`XF(5)` and `XF(6)`) (see `xfopt` for general information about database wrappers). The tabs of the property figure let you edit

- `DataBase` : general properties of datasets in `XF`.

- `Channel` : properties of each input/output pair in the current `XF`*i* dataset as well as rapid selection of which channel is displayed.
- `Axes` : detailed manipulations of axes displayed by `iiplot`.

The `d_iiplot` demo which will launch your browser at the HTML version of this page and initialize using `load sdt_id; iiplot`. Once there, try the following steps

- Type `iicom submagpha` to display a standard magnitude/phase plot. Open the `IIplot:sub commands` menu and see that you could have achieved the same thing using this pull-down menu. Note that you could also type the `submagpha` command in the text area near the button.
- Drag your mouse on the plot to select a region of interest and see how you directly zoom to this region. Double click on the same plot to go back to the initial zoom. On some platforms the double click is sensitive to speed and you may want to type the `i` key with the axis of interest active. An axis becomes active when you click on it. When, as here, you have more than one axis, the current axis button and `PlotType` pull-down menu are updated when a new axis becomes current.
- Open the `ContextMenu` associated with any axis (click anywhere in the axis using the right mouse button), select `Cursor`, and see how you have a vertical cursor giving information about data in the axis. To stop the cursor use a right click or press the `c` key.
- Notice the other things you can do with the `ContextMenu` : select lin or log scales, set axes title options, set pole line defaults, ...
- Open the `ContextMenu` associated with any line object (click on the line using the right mouse button), see how you can set line type, width, color ...
- Click on pole lines (vertical dotted lines) and FRFs and see how additional information on what you just clicked on is given. You can hide the info area by clicking on it.
- Type `iicom(';cax1;showmmi');` to display the MMIF in the lower plot. Go back to the phase, by making axis 1 active (click on it and the current axis button should show) and selecting `phase(w)` in the **axis type menu** (which is located just on the right of the current axis button).
- use the to scan trough different transfer functions. Note that you can also use the `+` or `-` keys when a drawing axis is active.
- Go the the `Channel` tab of the property figure and select more than one channel in the list. Note that you can also select channels from the command line using `iicom('ch 1 5')`.

- Make another data set using `IIxe=2*IIxf;` and overlay `IIxf` and `IIxe` using `iicom(';showabs;ch1;iixf on;IIxeOn')`. You could also achieve the same thing using the `IIplot:Variables` menu. Take a look at the `iiplot` reference section for a list of **global** variables used by the `iiplot` interface and remember that using the `clear` command on these variables only clears the link between the local and the global variable.
- 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`.
- continue the `d_iiplot` demo which shows a few other things.

After running through these steps, you should master the basics of the `iiplot` interface. To learn more, you should take time to see which commands are available by reading the *Reference* sections for `iicom` (general list of commands for plot manipulations), `iimouse` (mouse and key press support for SDT and non SDT figures), `iiplot` (standard plots derived from FRFs and test results that are supported).

3.2.4 Operational deflection shapes

Operational Deflection Shapes is a generic name used to designate the spatial relation of forced vibration measured at two or more sensors. Time responses of simultaneously acquired measurements, frequency responses to a possibly unknown input, transfer functions, transmissibilities, ... are example of ODS.

When displaying responses with `iiplot` and a test geometry with `feplot`, `iiplot` supports an ODS cursor. Run `demosdt('DemoGartteOds')` then open the context menu associated with any `iiplot` axis and select `ODS Cursor`. The deflection show in the `feplot` figure will change as you move the cursor in the `iiplot` window.

More generally, you can use `fecom Initdef` commands to display any shape as soon as you have a defined geometry and a response at DOFs. The `Deformations` tab of the `feplot` properties figure then lets you select deformations within a set.

```
cf=demosdt('DemoGartteOds')
cf.def=XF(1);
% or the low level call
% cf.def={XF(1).xf,XF(1).dof,XF(1).w}
fecom('curtab Deformation');
```

You can also display the actual measurements as arrows using

```
cf=demosdt('DemoGartteOds'); cf.def=XF(1);
cf.sens=XF(1).dof;fecom showarrow;
```

For a tutorial on the use of `feplot` see section 5.4.

3.3 Identification of modal properties

Identification is the process of estimating a parametric model (poles and modeshapes) that accurately represents measured data. The main algorithm proposed in the *SDT* is a frequency domain output error method that builds a model in the pole residue form (see section 2.6) through a tuning strategy. Key theoretical notions are pole/residue models, residual terms page 121, and the relation between residues and modeshapes (`cpx` page 35).

Section 3.3.1 gives a tutorial on the standard procedure. Theoretical details about the underlying algorithm are given in section 3.3.2. Section 3.3.3 addresses its typical shortcomings. Other methods implemented in the *SDT* but not considered as efficient are addressed in later sections.

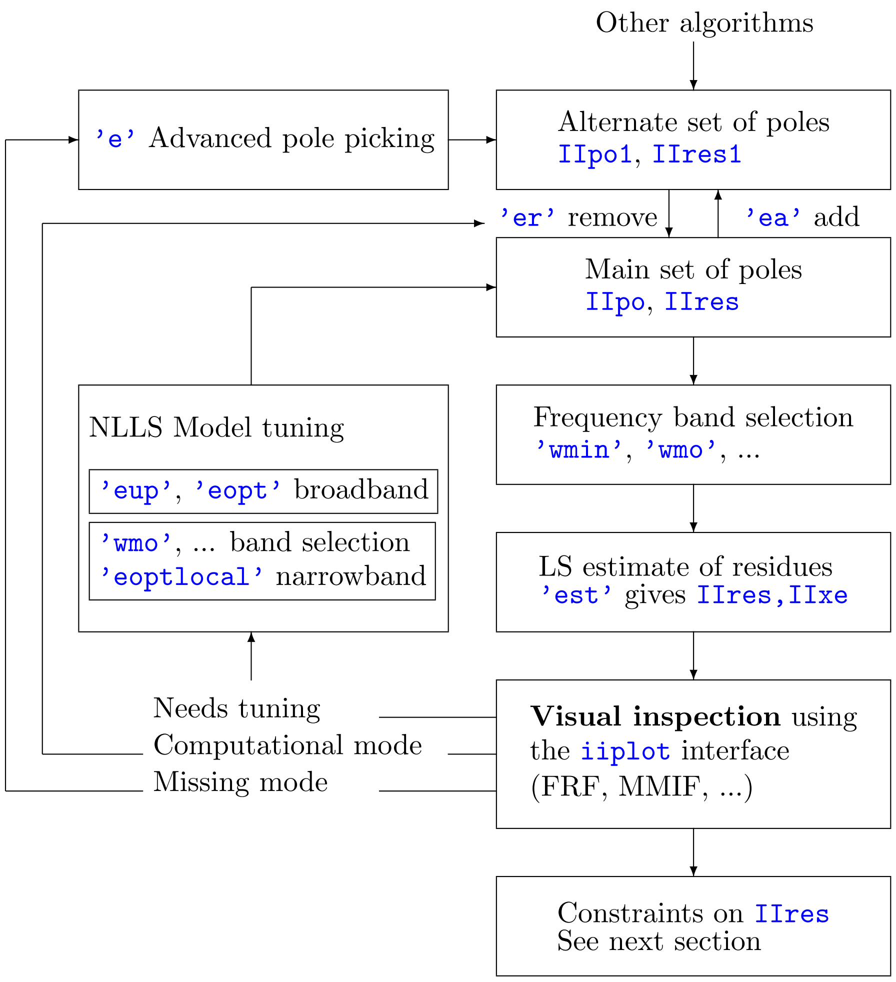
For the handling of MIMO tests, reciprocity, ... see section 3.4. The `gartid` script gives real data and an identification result for the GARTEUR example. The `demo_id` script analyses a simple identification example.

3.3.1 The `id_rc` procedure step by step

The `id_rc` identification method is based on an iterative refinement of the poles of the current model. Illustrated by the diagram below.

The main steps of the methodology are

- finding initial pole estimates (with the narrow band estimator, `idcom e` command), adding missed poles, removing computational poles (using the arrows between the main and alternate pole sets, `ea` and `er` commands)
- estimating residues and residual terms for a given set of poles (`est` command/button or direct call to `id_rc`)
- optimizing poles (and residues) of the current model using a broad or narrow band update (`eup`, `eopt`, `eoptlocal`, ... commands/buttons, with frequency band selection using the `wmin`, `wmo`, ... commands/buttons)


Other algorithms
'e' Advanced pole picking
Alternate set of poles IIpo1, IIres1
'er' remove
'ea' add
Main set of poles IIpo, IIres
NLLS Model tuning
'eup', 'eopt' broadband
'wmo', ... band selection
'eoptlocal' narrowband
Frequency band selection 'wmin', 'wmo', ...
LS estimate of residues 'est' gives IIres,IIxe
Needs tuning
Computational mode
Missing mode
Visual inspection using the iiplot interface (FRF, MMIF, ...)
Constraints on IIres
See next section

After verification of the Options tab of the idcom GUI figure, the Identification tab shown below gives you easy access to these steps (to open this figure, just run idcom from the MATLAB prompt). More details on how to proceed for each step are given below using data of the demo_id script.

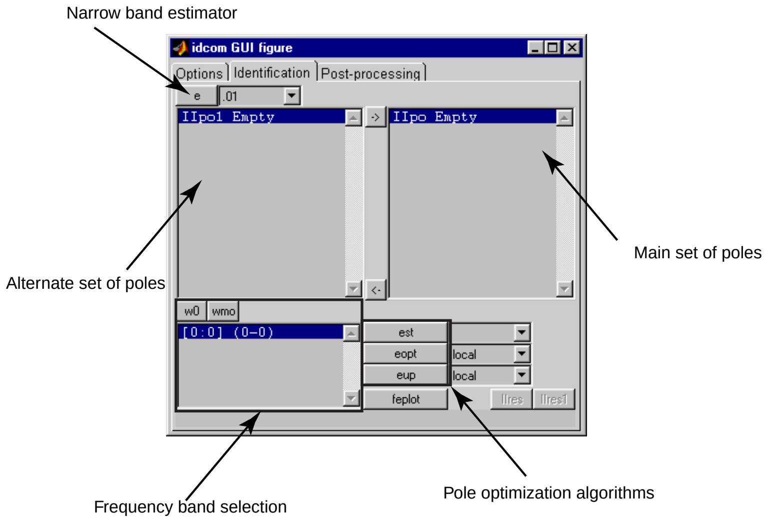


Figure 3.5: iicom interface.

The iteratively refined model is fully characterized by its poles (and the measured data). It is thus convenient to cut/paste the pole estimates into and out of a text editor (you can use the context menu of the main pole set to display this in the MATLAB command window). Saving the current pole set in a text file as the lines

```
IIpo = [   1.1298e+02   1.0009e-02
           1.6974e+02   1.2615e-02
           2.3190e+02   8.9411e-03];
```

gives you all you need to recreate an identified model (even if you delete the current one) but also lets you refine the model by adding the line corresponding to a pole that you might have omitted. The context menu associated with the pole set listboxes lets you easily generate this list.

1 finding initial pole estimates, adding missed poles, removing computational poles

Getting an initial estimate of the poles of the model is the first difficulty. Dynamic responses of structures, typically show lightly damped resonances. The easiest way to build an initial estimate of the poles is thus to use a sequence of narrow band single pole estimations near peaks of the response or minima of the Multivariate Mode Indicator function (use `iicom showmmi` and see `ii_mmif` for a full list of mode indicator functions).

The idcom e command (based on a call to the ii_poest function) lets you to indicate a frequency (with the mouse or by giving a frequency value) and seeks a single pole narrow band model near this frequency (the pole is stored in XF(6) (which points to IIpo1 and IIres1). Once the estimate found the iiplot drawing axes are updated to overlay XF(1) and XF(2).

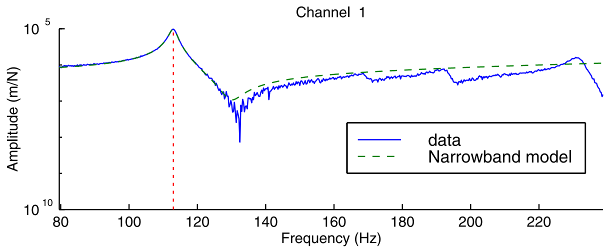


Figure 3.6: Pole estimation.

In the plot shown above the fit is clearly quite good. This can also be judged by the information displayed by ii_poest

```
LinLS:  1.563e-11, LogLS  8.974e-05, nw 10
mean(relE) 0.00, scatter 0.00
Found pole at 1.1299e+02   9.9994e-03
```

which indicates the linear and quadratic costs in the narrow frequency band used to find the pole, the number of points in the band, the mean relative error (norm of difference between test and model over norm of response which should be below 0.1), and the level of scatter (norm of real part over norm of residues, which should be small if the structure is close to having modal damping).

If you have a good fit and the pole differs from poles already in you current model, you can add the estimated pole (add poles in XF(6) to those in XF(5) IIpo) using the idcom ea command (or the associated button). If the fit is not appropriate you can change the number of selected points/bandwidth and/or the central frequency. In rare cases where the local pole estimate does not give appropriate results you can add a pole by just indicating its frequency (f command) or you can use the polynomial (id_poly), direct system parameter (id_dspi), or any other identification algorithm to find your poles. You can also consider the idcom find command which uses the MMIF to seek poles that are present in your data but not in IIpo.

In cases where you have added too many poles to your current model, the idcom er command then lets you remove certain poles.

This phase of the identification relies heavily on user involvement. You are expected to visualize the different FRFs (use the +/- buttons/keys), check different frequency bands (zoom with the mouse and use iicom w commands), use Bode, Nyquist, MMIF, etc. (see iicom show commands). The iiplot graphical user interface was designed to help you in this process and you should learn how to use it (you can get started in section 3.2).

2 estimating residues and residual terms

Once a model is created (you have estimated a set of poles), idcom est determines residues and displays the synthesized FRFs stored in XF(2) (which points to IIxe). A careful visualization of the data often leads to the discovery that some poles are missing from the initial model. The idcom e and ea commands can again be used to find initial estimates for the missing poles.

The need to add/remove poles is determined by careful examination of the match between the test data XF(1) and identified model XF(2). You should take the time to scan through different sensors, look at amplitude, phase, Nyquist, ...

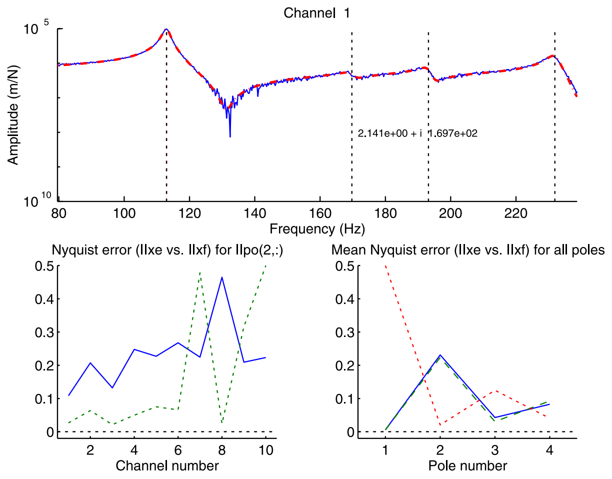


Figure 3.7: Pole estimation.

Quality and error plots are of particular interest. The quality plot (lower right, obtained with iicom showqual) gives an indication of the quality of the fit near each pole. Here pole 2 does not have a very good fit (relative error close to 0.2) but the response level (dotted line) is very small. The error plot (lower left, obtained with iicom showerr) shows the same information for the current pole and each transfer function (you change the current pole by clicking on pole lines in the top plot). Here it confirms that the relative Nyquist error is close to 0.2 for most channels. This clearly indicates the need to update this pole as detailed in the next section (in this example, the relative Nyquist error is close to 0.1 after updating).

3 updating poles of the current model using a broad or narrow frequency band update

The various procedures used to build the initial pole set (see step 1 above) tend to give good but not perfect approximations of the pole sets. In particular, they tend to optimize the model for a cost that differs from the broadband quadratic cost that is really of interest here and thus result in biased pole estimates.

It is therefore highly desirable to perform non-linear update of the poles in `XF(5)`. This update, which corresponds to a Non-Linear Least-Squares minimization, can be performed using the commands `idcom` `eup` (`id_rc` function) and `eopt` (`id_rcopt` function). The optimization problem is very non linear and non convex, good results are thus only found when improving results that are already acceptable (the result of phase 2 looks similar to the measured transfer function).

When using the `eup` command `id_rc` starts by reminding you of the currently selected options (global variable `IDopt`) for the type of residual corrections, model selected and, when needed, partial frequency range selected

```
 Low and high frequency mode correction
 Complex residue symmetric pole pattern
```

the algorithm then does a first estimation of residues and step directions and outputs

```
     mode#   dstep (%)     zeta      fstep (%)     freq
       1      10.000    1.0001e-02    -0.200    7.1043e+02
       2     -10.000    1.0001e-02     0.200    1.0569e+03
       3      10.000    1.0001e-02    -0.200    1.2176e+03
       4      10.000    1.0001e-02    -0.200    1.4587e+03
 Quadratic cost
    4.6869e-09
 Log-mag least-squares cost
    6.5772e+01
 how many more iterations? ([cr] for 1, 0 to exit) 30
```

which indicates the current pole positions, frequency and damping steps, as well as quadratic and logLS costs for the complete set of FRFs. These indications and particularly the way they improve after a few iterations should be used to determine when to stop iterating.

Here is a typical result after about 20 iterations

```
     mode#   dstep (%)     zeta      fstep (%)    freq
       1      -0.001    1.0005e-02     0.000    7.0993e+02
       2      -0.156    1.0481e-02    -0.001    1.0624e+03
       3      -0.020    9.9943e-03     0.000    1.2140e+03
       4      -0.039    1.0058e-02    -0.001    1.4560e+03
 Quadratic cost
  4.6869e-09 7.2729e-10 7.2741e-10 7.2686e-10 7.2697e-10
 Log-mag least-squares cost
  6.5772e+01 3.8229e+01 3.8270e+01 3.8232e+01 3.8196e+01
how many more iterations? ([cr] for 1, 0 to exit) 0
```

Satisfactory convergence can be judged by the convergence of the quadratic and logLS cost function values and the diminution of step sizes on the frequencies and damping ratios. In the example, the damping and frequency step-sizes of all the poles have been reduced by a factor higher than 50 to levels that are extremely low. Furthermore, both the quadratic and logLS costs have been significantly reduced (the leftmost value is the initial cost, the right most the current) and are now decreasing very slowly. These different factors indicate a good convergence and the model can be accepted (even though it is not exactly optimal).

The step size is divided by 2 every time the sign of the cost gradient changes (which generally corresponds passing over the optimal value). Thus, you need to have all (or at least most) steps divided by 8 for an acceptable convergence. Upon exit from `id_rc`, the `idcom eup` command displays an overlay of the measured data `XF(1)` and the model with updated poles `XF(2)`. As indicated before, you should use the error and quality plots to see if mode tuning is needed.

The optimization is performed in the selected frequency range (`idopt wmin` and `wmax` indices). It is often useful to select a narrow frequency band that contains a few poles and update these poles. When doing so, model poles whose frequency are not within the selected band should be kept but not updated (use the `euplocal` and `eoptlocal` commands). You can also update selected poles using the `'eup num` *i*`'` command (for example if you just added a pole that was previously missing).

`id_rc` (`eup` command) uses an ad-hoc optimization algorithm, that is not guaranteed to improve the result but has been found to be efficient during years of practice. `id_rcopt` (`eopt` command) uses a conjugate gradient algorithm which is guaranteed to improve the result but tends to get stuck at non optimal locations. You should use the `eopt` command when optimizing just one or two poles (for example using `eoptlocal` or `'eopt num i'` to optimize different poles sequentially).

In many practical applications the results obtained after this first set of iterations are incomplete. Quite often local poles will have been omitted and should now be appended to the current set of poles (going back to step 1). Furthermore some poles may be diverging (damping and/or frequency step not converging towards zero). This divergence will occur if you add too many poles (and these poles should be deleted) and may occur in cases with very closely spaced or local modes where the initial step or the errors linked to other poles change the local optimum for the pole significantly (in this case you should reset the pole to its initial value and restart the optimization).

Once a good complex residue model obtained, one often seeks models that verify other properties of minimality, reciprocity or represented in the second order mass, damping, stiffness form. These approximations are provided using the `id_rm` and `id_nor` algorithms as detailed in section 3.4.

3.3.2 Background theory

The `id_rc` algorithm (see [12][13]) seeks a non linear least squares approximation of the measured data

$$p_{\texttt{model}} = \arg\min \sum_{j,k,l=1}^{NS,NA,NW} \left(\alpha_{jk(\texttt{id})}(\omega_l, p) - \alpha_{jk(\texttt{test})}(\omega_l) \right)^2 \tag{3.1}$$

for models in the nominal pole/residue form (also often called partial fraction expansion [14])

$$[\alpha(s)] \;=\; \sum_{j\texttt{identified}} \left(\frac{[R_j]}{s-\lambda_j} + \frac{[\bar{R}_j]}{s-\bar{\lambda}_j} \right) + [E] + \frac{[F]}{s^2} = [\Phi(\lambda_j, s)]\,[R_j, E, F] \tag{3.2}$$

or its variants detailed under `res` page 37.

These models are linear functions of the residues and residual terms $[R_j, E, F]$ and non linear functions of the poles λ_j. The algorithm thus works in two stages with residues found as solution of a linear least-square problem and poles found through a non linear optimization.

The `id_rc` function (`idcom eup` command) uses an ad-hoc optimization where all poles are optimized simultaneously and steps and directions are found using gradient information. This algorithm is usually the most efficient when optimizing more than two poles simultaneously, but is not guaranteed to converge or even to improve the result.

The id_rcopt function (idcom eopt command) uses a gradient or conjugate gradient optimization. It is guaranteed to improve the result but tends to be very slow when optimizing poles that are not closely spaced (this is due to the fact that the optimization problem is non convex and poorly conditioned). The standard procedure for the use of these algorithms is described in section 3.3.1. Improved and more robust optimization strategies are still considered and will eventually find their way into the *SDT*.

3.3.3 When id_rc fails

This section gives a few examples of cases where a direct use of id_rc gave poor results. The proposed solutions may give you hints on what to look for if you encounter a particular problem.

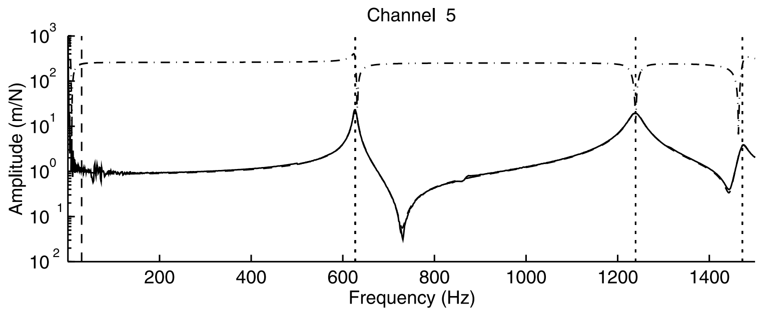


Figure 3.8: FRF estimation

In many cases frequencies of estimated FRFs go down to zero. The first few points in these estimates generally show very large errors which can be attributed to both signal processing errors and sensor limitations. The figure above, shows a typical case where the first few points are in error by orders of magnitude. Of two models with the same poles, the one that keeps the low frequency erroneous points (- — -) has a very large error while a model truncating the low frequency range (- - -) gives an extremely accurate fit of the data (—).

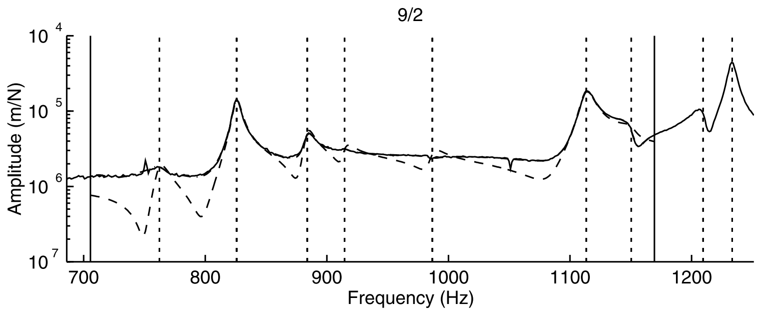


Figure 3.9: FRF xxx

The fact that appropriate residual terms are needed to obtain good results can have significant effects. The figure above shows a typical problem where the identification is performed in the band indicated by the two vertical solid lines. When using the 7 poles of the band, two modes above the selected band have a strong contribution so that the fit (- - -) is poor and shows peaks that are more apparent than needed (in the 900-1100 Hz range the FRF should look flat). When the two modes just above the band are introduced, the fit becomes almost perfect (- — -) (only visible near 750 Hz).

Keeping out of band modes when doing narrow band pole updates is thus quite important. You may also consider identifying groups of modes by doing sequential identifications for segments of your test frequency band [13].

The example below shows a related effect. A very significant improvement is obtained when doing the estimation while removing the first peak from the band. In this case the problem is actually linked to measurement noise on this first peak (the Nyquist plot shown in the lower left corner is far from the theoretical circle).

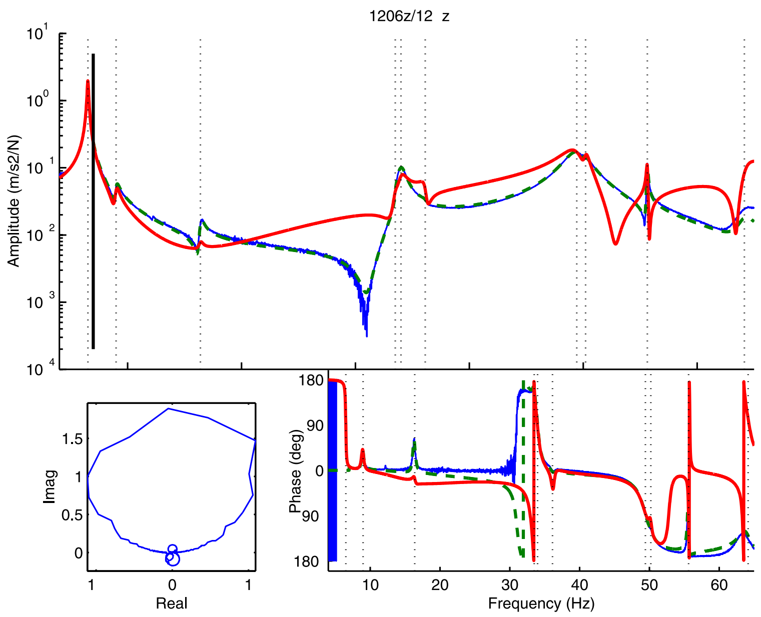


Figure 3.10: FRF xxx

Other problems are linked to poor test results. Typical sources of difficulties are

- mass loading (resonance shifts from FRF to FRF due to batch acquisition with displaced sensors between batches),
- leakage in the estimated FRFs,
- significant non-linearities (inducing non-symmetric resonances or resonance shifts for various excitation positions),
- medium frequency range behavior (the peaks of more than a few modes overlay significantly it can be very hard to separate the contributions of each mode even with MIMO excitation).

3.3.4 Direct system parameter identification algorithm

A class of identification algorithms makes a direct use of the second order parameterization. Although the general methodology introduced in previous sections was shown to be more efficient in general, the use of such algorithms may still be interesting for first-cut analyses. A major drawback of second order algorithms is that they fail to consider residual terms.

The algorithm proposed in id_dspi is derived from the direct system parameter identification algorithm introduced in Ref. [15]. Constraining the model to have the second-order form

$$\begin{array}{c} \left[-\omega^2 I + i\omega C_T + K_T\right]\{p(\omega)\} = [b_T]\{u(\omega)\} \\ \{y(\omega)\} = [c_T]\{p(\omega)\} \end{array} \tag{3.3}$$

it clearly appears that for known $[c_T]$, $\{y_T\}$, $\{u_T\}$ the system matrices $[C_T]$, $[K_T]$, and $[b_T]$ can be found as solutions of a linear least-squares problem.

For a given output frequency response $\{y_T\}$ =xout and input frequency content $\{u_T\}$ =xin, id_dspi determines an optimal output shape matrix $[c_T]$ and solves the least squares problem for $[C_T]$, $[K_T]$, and $[b_T]$. The results are given as a state-space model of the form

$$\begin{array}{c} \left\{\begin{array}{c}\dot q\\ \ddot q\end{array}\right\} = \left[\begin{array}{cc}0 & I\\ -K_T & -C_T\end{array}\right]\left\{\begin{array}{c}q\\ \dot q\end{array}\right\} + \left[\begin{array}{c}0\\ b_T\end{array}\right]\{u(t)\} \\ \{y(t)\} = [c_T \;\; 0]\left\{\begin{array}{c}q\\ \dot q\end{array}\right\} \end{array} \tag{3.4}$$

The frequency content of the input $\{u\}$ has a strong influence on the results obtained with id_dspi. Quite often it is efficient to use it as a weighting, rather than using a white input (column of ones) in which case the columns of $\{y\}$ are the transfer functions.

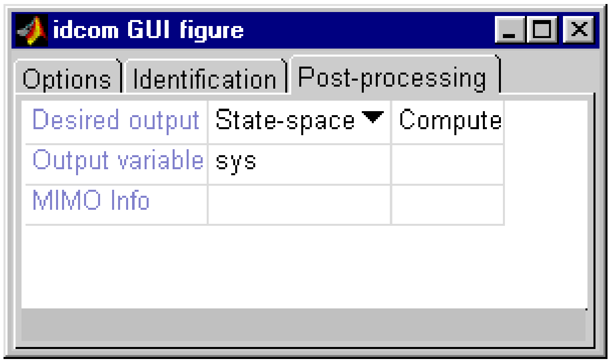
As no conditions are imposed on the reciprocity (symmetry) of the system matrices $[C_T]$ and $[K_T]$ and input/output shape matrices, the results of the algorithm are not directly related to the normal mode models identified by the general method. Results obtained by this method are thus not directly applicable to the prediction problems treated in section 3.4.2.

3.3.5 Orthogonal polynomial identification algorithm

Among other parameterizations used for identification purposes, polynomial representations of transfer functions (2.26) have been investigated in more detail. However for structures with a number of lightly damped poles, numerical conditioning is often a problem. These problems are less acute when using orthogonal polynomials as proposed in Ref. [16]. This orthogonal polynomial method is implemented in id_poly, which is meant as a flexible tool for initial analyses of frequency response functions. This function is available as idcom poly command.

3.4 MIMO, Reciprocity, State-space, ...

The pole/residue representation is often not the desired format. Access to transformations is provided by the post-processing tab in the `idcom` properties figure. There you can select the desired output format and the name of the variable in the base MATLAB workspace you want the results to be stored in.



Desired output	State-space	Compute
Output variable	sys	
MIMO Info		

Figure 3.11: idcom interface

The `id_rm` algorithm is used for the creation of minimal and/or reciprocal pole/residue models (from the command line use `sys=id_rm(XF(5))`). For the extra step of state-space model creation use `sys=res2ss(XF(5))`.
`nor=res2nor(XF(5))` or `nor=id_nor(XF(5))` allow transformations to the normal mode form. Finally direct conversions to other formats are given by `struct=res2xf(XF(5),IIw)` and `[num,den]=res2xf(XF(5))`.

These calls are illustrated in `demo_id`.

3.4.1 Multiplicity (minimal state-space model)

Theory

As mentioned under `res` page 37 , the residue matrix of a mode can be written as the product of the input and output shape matrices, so that the modal contribution takes the form

$$\frac{R_j}{s-\lambda_j} = \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j} \tag{3.5}$$

For a single mode, the product $\{c\psi_j\}\left\{\psi_j^T b\right\}$ has rank 1. Thus for a truly MIMO test (with more than one input and output), the residue matrix found by `id_rc` usually has full rank and cannot be written as shown in (3.5). In some cases, two poles of a structure are so close that they can be considered as a multiple pole $\lambda_j = \lambda_{j+1}$, so that

$$\frac{R_j}{s-\lambda_j} = \frac{\{c\psi_j\}\left\{\psi_j^T b\right\} + \{c\psi_{j+1}\}\left\{\psi_{j+1}^T b\right\}}{s-\lambda_j} \tag{3.6}$$

In such cases, the residue matrix $[R_j]$ has rank two. **Minimality** (i.e. rank constraint on the residue matrix) is achieved by computing, for each mode, the singular value decomposition of the residue matrix $R_j = U\Sigma V^T$. By definition of the singular value decomposition

$$\left[\tilde{R}_j\right]_{NS\times NA} = \{U_1\}_{NS\times 1}\,\sigma_1\,\{V_1\}_{NA\times 1}^T \tag{3.7}$$

is the best rank 1 approximation (in the matrix norm sense) of R_j. Furthermore, the ratio σ_2/σ_1 is a measure of the relative error made by retaining only the first dyad. This ratio gives, for MIMO tests, an indication of the coherence of estimated modeshapes and occasionally an indication of the pole multiplicity if two poles are sufficiently close to be considered as identical (see the example below).

Minimal pole/residue models are directly linked to a state-space model of the form

$$\begin{array}{c}\left(s\,[I]_{2N\times 2N} - \left[\backslash\lambda_{j\backslash}\right]\right)\{\eta\} = \left[\psi^T b\right]\{u\} \\ \{y\} = [c\psi]\,\{\eta\}\end{array} \tag{3.8}$$

which can then be transformed to a real valued state-space model (see `res2ss`) or a second order normal mode model (see section 3.4.3).

Practice `id_rm` builds a rank constrained approximation of the residue matrix associated to each pole. When not enforcing reciprocity, the output of the call

```
demosdt('demo demo_id')
XF(5).idopt.nsna=[5 2];  XF(5).idopt.reci='no';
RES = id_rm(XF(5),[1 2 1 1]);
% or low level call
[pb,cp,new_res]=id_rm(IIres,IIpo,XF(5).idopt,[1 2 1 1]);
```

returns an output that has has the form

```
The system has 5 sensors and 2 actuators
FRF 7 (actuator 2 sensor 2) is collocated
     Po #    freq    mul     Ratio of sing. val. to max
      1    7.10e+02  2  :   0.3000 k  0.0029
```

```
   2    9.10e+02  1  :   0.1000    0.0002
   3    1.20e+03  1  :   0.0050    0.0001
   4    1.50e+03  1  :   0.0300    0.0000
```

where the first three columns indicate pole number, frequency and retained multiplicity and the following give an indication of the difference between the full rank residue matrix and the rank constrained one (the singular value ratio should be much smaller than 1).

In the result show above, pole 1 is close to being rank 2 since the difference between the full order residue matrix and a rank 1 approximation is of the order of 30% while the difference with a rank 2 approximation is only near 0.2%.

The fact that a rank 1 approximation is not very good can be linked to actual multiplicity but more often indicates poor identification or incoherent data. For poor identification the associated pole should be updated as shown in section 3.3. For incoherent data (for example modes slightly modified due to changing shakers during sequential SIMO tests), one should perform separate identifications for each set of coherent measurements. The rank constrained approximation can then be a way to reconcile the various results obtained for each identification.

If the rank of the residue matrix is truly linked to pole multiplicity, one should try to update the identification in the vicinity of the pole: select a narrow frequency range near this pole, then create and optimize a two or more pole model as shown section 3.3.1. True modal multiplicity being almost impossible to design into a physical structure, it is generally possible to resolve such problems. Keeping multiple poles should thus only remain an intermediate step when not having the time to do better.

3.4.2 Reciprocal models of structures

Theory

In many cases, the structures tested are assumed to be *reciprocal* (the transfers force at A/response at B and force at B/response at A are equal) and one wants to build a reciprocal model. For modal contributions of the form (3.5), reciprocity corresponds to the equality of collocated input and output shape matrices

$$\left([c_{\texttt{col}}]\,\{\psi_j\}\right)^T = \{\psi_j\}^T\,[b_{\texttt{col}}] \tag{3.9}$$

For reciprocal structures, the residue matrix associated to collocated FRFs should be symmetric. `id_rm` thus starts computing the symmetric part of the collocated residues $\hat{R}_{j\texttt{col}} = \left(R_{j\texttt{col}} + R_{j\texttt{col}}^T\right)/2$. This matrix being symmetric, its singular value decomposition is given by $\hat{R}_{j\texttt{col}} = U_{\texttt{col}}\Sigma_{\texttt{col}}V_{\texttt{col}}^T$ which leads to the reciprocal input and output shape matrices

$$\{c_{\texttt{col}}\psi_j\} = \left\{\psi_j^T b_{\texttt{col}}\right\}^T = \sqrt{\sigma_{1\texttt{col}}}\,\{U_{1\texttt{col}}\} \tag{3.10}$$

Typically, there are many more sensors than inputs. The decomposition (3.10) is thus only used to determine the collocated input shape matrices and the output shape matrices at all sensors are found as solution of a least square problem $\{c\psi_j\} = [R_j]\left\{\psi_j^T b_{\texttt{col}}\right\}^+$ which does require that all inputs have a collocated sensor.

Reciprocity provides scaled input and output shape matrices. This scaling is the same as that obtained with the analytical scaling condition (2.19). The interest of using reciprocal models is to predict non measured transfer functions.

Practice When collocated transfer functions are declared and `IDopt.Reciprocity='1 FRF'` or `MIMO`, `id_rm` seeks a minimal and reciprocal approximation to the model. For the call

```
demosdt('demo demo_id')
XF(5).idopt.nsna=[5 2]; XF(5).idopt.Col=[1 7];
XF(5).idopt.reci='mimo';
RES = id_rm(XF(5),[1 1 1 1]);
XF(3)=res2xf(RES,IIw);iicom('IIxhOn')
% or low level call
[pb,cp,new_res]=id_rm(IIres,IIpo,XF(5).idopt,[1 1 1 1]);
IIxh = res2xf(new_res,IIpo,IIw,IDopt); iicom('IIxhOn')
```

`id_rm` shows information of the form

```
 The system has 5 sensors and 2 actuators
 FRF  1 (actuator  1 sensor  1) is collocated
 FRF  7 (actuator  2 sensor  2) is collocated
Reciprocal MIMO system
 Po#    freq    mul     sym.    rel.e.
   1    1.13e+02  1 :   0.0001   0.0002
   2    1.70e+02  1 :   0.0020   0.0040
   3    1.93e+02  1 :   0.0003   0.0005
   4    2.32e+02  1 :   0.0022   0.0044
```

where the output indicates the number of sensors and actuators, the collocated FRFs, the fact the resulting model will enforce MIMO reciprocity, and details the accuracy achieved for each mode.

The algorithm first enforces symmetry on the declared collocated transfer functions the symmetry error sym. shows how asymmetric the original residue matrices where. If for a given mode this number is not close to zero, the mode is poorly identified or the data is far from verifying reciprocity and building a reciprocal model makes no sense.

The algorithm then seeks a rank constrained approximation, the relative error number rel. e. shows how good an approximation of the initial residue matrix the final result is. If this number is larger than .1, you should go back to identifying a minimal but non reciprocal model, determine the actual multiplicity, and update the pole, if it is not very well identified, or verify that your data is really reciprocal.

You can check the accuracy of FRF predicted with the associated model using the synthesized FRFs (IIxh/XF(3) in the example above). An alternate FRF generation call would be

```
[a,b,c,d]=res2ss(res,po,idopt);
IIxh=qbode(a,b,c,d,IIw*2*pi);
```

This more expensive computationally, but state-space models are particularly useful for coupled system analysis and control synthesis.

You can also use reciprocal models to predict the response of untested transfer functions. For example the response associated to a shaker placed at the uind sensor (not a collocated one) can be computed using

```
demosdt('demo demo_id')
[psib,cpsi]=id_rm(IIres,IIpo,IDopt,[1 1 1 1]);
uind=3; res_u = (cpsi*diag(cpsi(uind,:))).';
IIxe=res2xf(res_u,IIpo,IIw,IDopt); iiplot
```

You should note that the res_u model does not contain any residual terms, since reciprocity does not give any information on those. Good predictions of unmeasured transfers are thus limited to cases where residual terms can be neglected (which is very hard to know *a priori*).

3.4.3 Normal mode form

Modal damping assumption

While the most accurate viscous damping models are obtained with a full damping matrix Γ (supported by psi2nor and id_nor as detailed in the next section), **modal**

damping (where Γ is assumed diagonal which is valid assumption when (3.15) is verified) is used in most industrial applications and is directly supported by `id_rc`, `id_rm` and `res2nor`. The use of this functionality is demonstrated in `demo_id`.

For a modally damped model (diagonal modal damping matrix Γ), the normal mode model (2.4) can be rewritten in a rational fraction form (with truncation and residual terms)

$$[\alpha(s)] = \sum_{j=1}^{NM} \frac{\{c\phi_j\}\left\{b^T\phi_j\right\}^T}{s^2+2\zeta_j\omega_j s+\omega_j^2} + [E] + \frac{[F]}{s^2} = \sum_{j=1}^{NM} \frac{[T_j]_{NS\times NA}}{s^2+2\zeta_j\omega_j s+\omega_j^2} + E(s) \quad (3.11)$$

This parameterization, called *normal mode residue form*, has a symmetric pole pattern and is supported by various functions (`id_rc`, `id_rm`, `res2xf` , ...) through the use of the option `IDopt.Fit='Normal'`. As for the complex residues (2.25), the normal mode residue matrix given by `id_rc` and used by other functions is stacked using one row for each pole or asymptotic correction term and, as the FRFs (see the `xf` format), a column for each SISO transfer function (stacking NS columns for actuator 1, then NS columns for actuator 2, etc.)

Assuming that the constraint of proportional damping is valid, the identified residue matrix T_j is directly related to the true normal modes

$$[T_j] = \{c\phi_j\}\left\{\phi_j^T b\right\} \quad (3.12)$$

and the dyadic decomposition of the residue matrix can be used as in the complex mode case (see section 3.4.1 and the function `id_rm`) to obtain a minimal and/or reciprocal models (as well as scaled input and output shape matrices).

The scaling implied by equations (3.11) and (3.12) and used in the functions of the *Toolbox* is consistent with the assumption of unit mass normalization of the normal modes (see details under `nor` page 28). This remains true even for multiple modes. A result rarely obtained by other methods.

When a complex mode identification has been performed (`IDopt.Fit='Complex'` or `'Posit'`), the function `res2nor` also provides a simple approximation of the complex residue model by a normal mode residue model.

Non proportional damping assumption

Theory The complex modes of a minimal/reciprocal model are related to the mass / damping / stiffness matrices by (see Ref. [5])

$$M = \left(\tilde\psi\Lambda\tilde\psi^T\right)^{-1}, \quad C = -M\tilde\psi\Lambda^2\tilde\psi^T M, \quad \text{and} \quad K = \left(\tilde\psi\Lambda^{-1}\tilde\psi^T\right)^{-1} \quad (3.13)$$

if and only if the complex modes are also proper. That is, they verify verify

$$\sum_{j=1}^{2N} \left\{\tilde{\psi}_j\right\} \left\{\tilde{\psi}_j\right\}^T = \left[\tilde{\psi}\right]_{N\times 2N} \left[\tilde{\psi}\right]^T_{N\times 2N} = [0]_{N\times N} \tag{3.14}$$

The transformation `id_nor` is thus done in two stages. `id_rm` is used to find a minimal and reciprocal approximation of the identified residue model of the form (3.8). `psi2nor` then determines c and $\tilde{\psi}$ such that the $\tilde{\psi}$ verify the condition (3.14) and $c\tilde{\psi}$ is "optimally" close to the $c\psi$ resulting from `id_rm`. Using the complex modes $\tilde{\psi}$ and the identified poles λ, the matrices are then computed and the model transformed to the standard normal mode form with no further approximation.

The possibility to perform the transformation is based on the fact that the considered group of modes is not significantly coupled to other modes by damping [5]. Groups of modes which can be approximated by a second order non proportionally damped model can be easily detected using the frequency separation criterion which must be verified between modes j in the group and modes k outside the group

$$\frac{\zeta_j\omega_j\zeta_k\omega_k{}^2}{\omega_j\omega_k} \ll 1 \tag{3.15}$$

If there does not exist a normal mode model that has complex modes close to the identification result $c\psi$, the algorithm may not work. This will happen in particular if $c\psi\Lambda\psi^T c^T = cM^{-1}c^T$ does not have NQ positive eigenvalues (estimated mass not positive definite).

Practice

The use of this functionality is demonstrated in `demo_id`. For comparisons with undamped FE models, it is essential to obtain estimates of normal modes. The most accurate results are obtained using a non-proportionally damped normal mode model

```
[om,ga,pb,cp] = id_nor(IIres,IIpo,IDopt);
```

but approximate transformations based on the assumption of proportional damping can be obtained with `res2nor`. This is particularly useful if the identification is not good enough to build the minimal and reciprocal model used by `id_nor`. In such cases you can also consider using `id_rc` with the assumption of proportional damping which directly identifies normal modes (see more details in section 3.4.3).

The FRFs associated to the normal mode model can be computed using

```
IIxe = nor2xf(om,ga,pb,cp,IIw*2*pi);
```

Scaling problems are often encountered when using the reciprocity to condition to scale the complex modes in `id_rm`. The function `id_nor` allows an optimization of collocated residues based on a comparison of the identified residues and those linked to the normal mode model. You should be aware that `id_nor` only works on very good identification results, so that trying it without spending the time to go through the pole update phase of `id_rc` makes little sense.

The normal mode input `pb` and output `cp` matrices correspond to those of an analytical model with mass normalized modes. They can be compared (`ii_mac`, `ii_comac`) or combined (`fe_exp`) with analytical models and the modal frequencies `om` and damping matrix `ga` can be used for predictions (see more details in section 4.4).

The identified models only take some complex modes into account. Other modes and residual terms are here represented by the $E(s)$ term and must be retained (as in the example treated in `demo_id`). Second order models are said to be complete when $E(s)$ can be neglected [17].

The `id_nor` and `res2nor` algorithms only seek approximations the modes. For FRF predictions one will often have to add the residual terms. The figure below (taken from `demo_id`) shows an example where including residual terms tremendously improves the prediction.

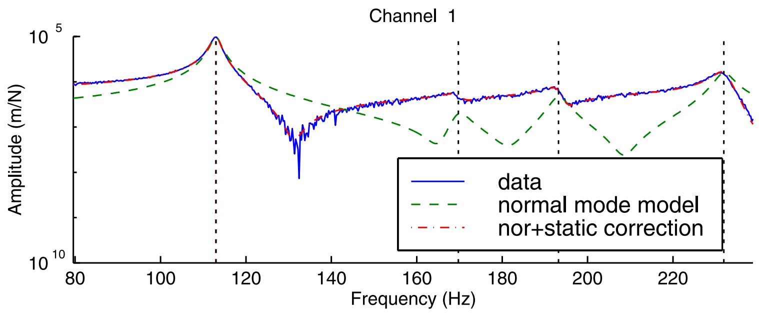


Figure 3.12: FRF xx

The residual mass and flexibility contributions of a four poles model can be found using

```
IIxh = IIxe + res2xf(IIres,IIpo,IIw,IDopt,[5 6]);
```

or equivalently with

```
[new_res,IIpo] = id_nor(IIres,IIpo,IDopt)
IIxh = res2xf(new_res,IIpo,IIw,IDopt)
```

4

Test/analysis correlation tutorial

Modal testing differs from system identification in the fact that responses are measured at a number of sensors which have a spatial distribution which allows the visualization of the measured motion. Visualization is key for a proper assessment of the quality of an experimental result. One typically considers three levels of models.

- Input/output models are defined at sensors. In the figure, one represents these sensors as arrows corresponding to the line of sight measurements of a laser vibrometer. Input/output models are the direct result of the identification procedure described in chapter 3.
- Wire frame models are used to visualize test results. They are an essential verification tool for the experimentalist. Designing a test well, includes making sure that the wire frame representation is sufficiently detailed to give the experimentalist a good understanding of the measured motion. With non-triaxial measurements, a significant difficulty is to handle the perception of motion assumed to be zero.
- Finite element models are used for test/analysis correlation. In most industrial applications, test and FEM nodes are not coincident so that special care must be taken when predicting FEM motion at test nodes/sensors (shape observation) or estimating test motion at FEM DOFs (shape expansion).

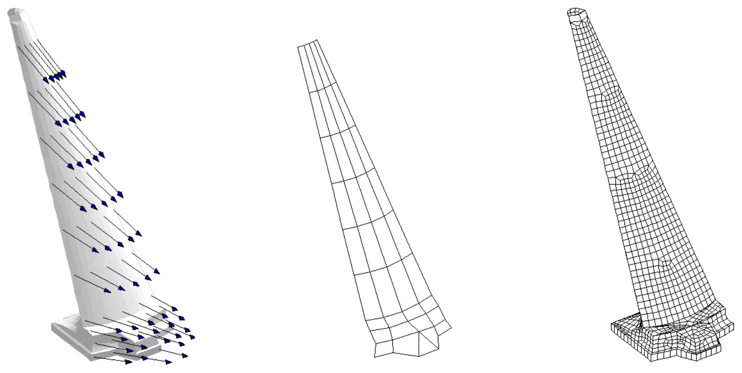


Figure 4.1: FE and wire-frame models

The tools for the declaration of the wire-frame model and of sensor setups are detailed in section 3.1. Topology correlation and sensor/shaker placement tools are details in section 4.1. A summary of general tools used to compare sets of shapes is made in section 4.2. Shape expansion, which deals with the transformations between the wire-frame and FE models, is introduced in section 4.3. The results of correlation can be used for hybrid models combining experimental and analytical results (see section 4.4) or for finite element model updating (see section 6.4).

4.1 Topology correlation and test preparation

Topology correlation is the phase where one correlates test and model geometrical and sensor/shaker configurations. Most of this effort is handled by `fe_sens` with some use of `femesh`.

As described in the following sections the three important phases of topology correlation are

- combining test and FEM model including coordinate system definition for the test nodes if there is a coordinate system mismatch,
- building of an observation matrix allowing the prediction of measurements based on FEM deformations,
- sensor and shaker placement.

4.1.1 Combining models

Given a FEM model (see section 5.1) and a test wire-frame model (see section 3.1.1), the first step is make sure that the wire-frame is indeed declared as a wire frame and to combine these models. This is easily done with the `femesh addtest` command as follows (see also the `gartte` demo)

```
FEM =demosdt('demo gartfe');
TEST=demosdt('demo gartte');
TEST.Elt=feutil('setgroupall egid-1',TEST); % declare as wire-frame
sens=feutil('addtest',FEM,TEST);
feutil('infoelt',sens) % put the combined model in sens
cf=feplot;cf.model=sens;fecom('showlinks');
```

The `FEM` model must describe nodes, elements and DOFs. The test wire frame `TEST` must describe nodes and lines/elements.

In many practical applications, the coordinate systems for test and FEM differ. `fe_sens` supports the use of a local coordinate system for test nodes with the `basis` command. For an example use

```
sens=demosdt('demo gartte basis');
cf=feplot;cf.model=sens;
pause
sens=fe_sens('basis estimate',sens);
cf.model=sens
fe_sens('tdof',sens)
fecom('view3')
```

Note that sensors defined using a `sens.tdof` DOF definition vector use the response coordinate system information given in column 3 of `sens.Node` while the 5 column format gives sensor directions in the global FEM coordinate system. In the example above, position and displacement coordinate systems for test nodes are set to 100. Thus the sensor `1011.02` (`sens.tdof(1)`) is a measurement in FEM direction z.

4.1.2 Observation matrix for a sensor configuration

`fe_sens` only assumes that measurements are linearly related to DOFs by an observation equation

$$\{y(t)\}_{NS\times 1} = [c]_{NS\times N} \ \{q(t)\}_{N\times 1} \tag{4.1}$$

This actually allows you to deal with non translation sensors sensors (rotation, strain, or any measurement linearly related to finite element DOF displacement). For a theoretical discussion of the methods discussed here see [18].

`fe_sens` actually supports mixed translation/rotation sensors using a DOF definition vector in the `sens.tdof` field, or translation sensors in arbitrary directions using the 5 column format discussed in section 3.1.2. For scanning laser vibrometer tests consider using the `fe_sens laser` command to define `sens.tdof`.

For topology correlation, the sensor configuration must be stored in the `sens.tdof` field and active FEM DOFs must be declared in `sens.DOF`. If you do not have your analysis modeshapes yet, you can use `sens.DOF=feutil('getdof',sens.DOF)`. With these fields and a combined test/FEM model you can estimate test node motion from FEM results. Available interpolations are

`near`	defines the projection based on a nearest node match.
`rigid`	defines the projection based on a nearest node match but assumes a rigid body link between the DOFs of the FE model and the test DOFs to obtain the DOF definition vector `adof` describing DOFs used for FEM results.
`arigid`	is a variant of the rigid link that estimates rotations based on translations of other nodes. This interpolation is more accurate than `rigid` for solid elements (since they don't have rotational DOFs) and shells (since the value of drilling rotations is often poorly related to the physical rotation of a small segment).

Since the nearest nodes is not necessarily the linked to the elements on which the sensor is glued, you may want to ensure that the observation matrices created by these commands only use nodes associated to a subset of elements. You can use *FEMNodeSelectors* and *TestNodeSelectors* arguments to force matching in particular node subsets. This is illustrated below in forcing the interpolation of test node 1206 to use FEM nodes in the plane where it is glued.

```
sens=demosdt('demo gartte cor');
sens=fe_sens('arigid',sens); % initial estimate
% sens=fe_sens('arigid',sens,'TestNodeSelectors','FEMNodeSelectors');
sens=fe_sens('rigid',sens,'nodeid 1206','z>.15');
% modify link to 1206
fe_sens('plotlinks',sens);fecom('textnode',1206)
fe_sens('info',sens)
```

At each point, you can see which interpolations you are using with `fe_sens('info',sens)`. **Note** that when defining test nodes in a local basis, the node selection commands are applied in the global coordinate system.

The interpolations are stored in the `sens.cta` field. With that information you can predict the response of the FEM model at test nodes. For example

```
[sens,def]=demosdt('demo gartte cor');
sens=fe_sens('rigid',sens); % initial estimate
cf=feplot; cf.model=sens; fecom('showtest')
cf.def={sens.cta*def.def,sens.tdof,def.data}
fecom(';undefline;scd.5;ch7')
```

The most common source of problems with the topology correlation commands is the use of models with nodes not attached to any element. All test nodes must be linked to the wire-frame model and all FEM Nodes must be linked to an element. A standard procedure to force `fe_sens` to consider additional nodes is to declare them in a group of mass elements as show in the example below

```
sens=demosdt('demo gartte cor');
% Add test nodes to wire-frame
elt=feutil('objectmass egid-1',[1012 1112]);
sens.Elt(end+[1:size(elt,1)],1:size(elt,2))=elt;
% Add test nodes to FEM model
elt=feutil('objectmass egid0',[62 47]);
sens.Elt(end+[1:size(elt,1)],1:size(elt,2))=elt;
feutil('infoelt',sens)
```

4.1.3 Sensor/shaker placement

In cases where an analytical model of a structure is available before the modal test, it is good practice to use the model to design the sensor/shaker configuration.

Typical objectives for sensor placement are

- Wire frame representations resulting from the placement should allow a good visualization of test results without expansion. Achieving this objective, enhances the ability of people doing the test to diagnose problems with the test, which is obviously very desirable.
- Seen at sensors, it is desirable that modes *look* different. This is measured by the condition number of $[c\phi]^T [c\phi]$ (modeshape independence, see [19]) or by the magnitude of off-diagonal terms in the auto-MAC matrix (this measures orthogonality). Both independence and orthogonality are strongly related.
- sensitivity of measured modeshape to a particular physical parameter (parameter visibility)

Sensor placement capabilities are accessed using the `fe_sens` function as illustrated in the `gartsens` demo. This function supports the effective independence [19] and maximum sequence algorithms which seek to provide good placement in terms of modeshape independence.

It is always good practice to verify the orthogonality of FEM modes at sensors using the auto-MAC (whose off-diagonal terms should typically be below 0.1)

```
cphi = fe_c(mdof,sdof)*mode; ii_mac('cpa',cphi,'mac auto plot')
```

For shaker placement, you typically want to make sure that

- you excite a set of target modes,
- or will have a combination of simultaneous loads that excites a particular mode and not other nearby modes.

The placement based on the first objective is easily achieved looking at the minimum controlability, the second uses the Multivariate Mode Indicator function (see `ii_mmif`). Appropriate calls are illustrated in the `gartsens` demo.

4.2 Test/analysis correlation

Correlation criteria seek to analyse the similarity and differences between two sets of results. Usual applications are the correlation of test and analysis results and the comparison of various analysis results.

Ideally, correlation criteria should quantify the ability of two models to make the same predictions. Since, the predictions of interest for a particular model can rarely be pinpointed precisely, one has to use general qualities and select, from a list of possible criterion, the ones that can be computed and do a good enough job for the intended purpose.

4.2.1 Shape based criteria

The `ii_mac` interface implements a number of correlation criteria. You should at least learn about the Modal Assurance Criterion (MAC) and Pseudo Orthogonality Checks (POC). These are very popular and should be used first. Other criteria should be used to get more insight when you don't have the desired answer or to make sure that your answer is really foolproof.

Again, *there is no best choice for a correlation criterion* unless you are very specific as to what you are trying to do with your model. Since that rarely happens, you should know the possibilities and stick to what is good enough for the job.

The following table gives a list of criteria implemented in the `ii_mac` interface.

`MAC`	Modal Assurance Criterion (9.7). The most popular criterion for correlating vectors. Insensitive to vector scaling. Sensitive to sensor selection and level of response at each sensor. Main limitation : can give very misleading results without warning. Main advantage : can be used in all cases. A MAC criterion applied to frequency responses is called FRAC.
`POC`	Pseudo Orthogonality Checks (9.9). Required in some industries for model validation. This criterion is only defined for modes since other shapes do verify orthogonality conditions. Its scaled insensitive version (9.8) corresponds to a mass weighted MAC and is implemented as the `MAC M` commands. Main limitation : requires the definition of a mass associated with the known modeshape components. Main advantage : gives a much more reliable indication of correlation than the MAC.
`Error`	Modeshape pairing (based on the MAC or MAC-M) and relative frequency error and MAC correlation.
`Rel`	Relative error (9.10). Insensitive to scale when using the modal scale factor. Extremely accurate criterion but does not tell much when correlation poor.
`COMAC`	Coordinate Modal Assurance Criteria (three variants implemented in `ii_mac`) compare sets of vectors to analyze which sensors lead poor correlation. Main limitation : does not systematically give good indications. Main advantage : a very fast tool giving more insight into the reasons of poor correlation.
`MACCO`	What if analysis, where coordinates are sequentially eliminated from the MAC. Slower but more precise than COMAC.

4.2.2 Energy based criteria

The criteria that make the most mechanical sense are derived from the equilibrium equations. For example, modes are defined by the eigenvalue problem (6.4). Thus the dynamic residual

$$\left\{\hat{R}_j\right\} = \left[K - \omega_{j\mathtt{id}}^2 M\right] \{\phi_{\mathtt{id}j}\} \tag{4.2}$$

should be close to zero. A similar residual (4.6) can be defined for FRFs.

The Euclidean norm of the dynamic residual has often been considered, but it tends to be a rather poor choice for models mixing translations and rotations or having very different levels of response in different parts of the structure.

To go to an energy based norm, the easiest is to build a displacement residual

$$\{R_j\} = \left[\hat{K}\right]^{-1} \left[K - \omega_{j\mathtt{id}}^2 M\right] \{\phi_{\mathtt{id}j}\} \tag{4.3}$$

and to use the strain $|\tilde{R}_j|_K = \tilde{R}_j^T K \tilde{R}_j$ or kinetic $|\tilde{R}_j|_M = \tilde{R}_j^T M \tilde{R}_j$ energy norms for comparison.

Note that $\left[\hat{K}\right]$ need only be a reference stiffness that appropriately captures the system behavior. Thus for cases with rigid body modes, a pseudo-inverse of the stiffness (see section 6.1.4), or a mass shifted stiffness can be used. The displacement residual $\tilde{R}_j$ is sometimes called error in constitutive law (for reasons that have nothing to do with structural dynamics).

This approach is illustrated in the `gartco` demo and used for MDRE in `fe_exp`. While much more powerful than methods implemented in `ii_mac`, the development of standard energy based criteria is still a fairly open research topic.

4.2.3 Correlation of FRFs

Comparisons of frequency response functions is performed for both identification and finite element updating purposes.

The quadratic cost function associated with the Euclidean norm

$$J_{ij}(\Omega) = \sum_{ij \ \text{measured}, k\in\Omega} |\hat{H}_{ij}(s_k) - H_{ij}(s_k)|^2 \tag{4.4}$$

is the most common comparison criterion. The main reason to use it is that it leads to linear least-squares problem for which there are numerically efficient solvers. (`id_rc` uses this cost function for this reason).

The quadratic cost corresponds to an additive description of the error on the transfer functions and, in the absence of weighting, it is mostly sensitive to errors in regions with high levels of response.

The log least-squares cost, defined by

$$J_{ij}(\Omega) = \sum_{ij \ \text{measured}, k\in\Omega} |log \left|\frac{\hat{H}_{ij}(s_k)}{H_{ij}(s_k)}\right||^2 \tag{4.5}$$

uses a multiplicative description of the error and is as sensitive to resonances than to anti-resonances. While the use of a non-linear cost function results in much higher computational costs, this cost tends to be much better at distinguishing physically close dynamic systems than the quadratic cost (except when the difference is very small which is why the quadratic cost can be used in identification phases).

The utility function ii_cost computes these two costs for two sets of FRFs xf1 and xf2 (obtained through test and FE prediction using nor2xf for example). The evaluation of these costs provides a quick and efficient way to compare sets of MIMO FRF and is used in identification and model update algorithms.

Note that you might also consider the complex log of the transfer functions which would give a simple mechanism to take phase errors into account (this might become important for extremely accurate identification sometimes needed for control synthesis).

If the response at a given frequency can be expanded to the full finite element DOF set, you should consider an energy criterion based on the dynamic residual in displacement, which in this case takes the form

$$\{R_j\} = \left[\hat{K}\right]^{-1} \left[[Z(\omega)] \{q_{ex}(\omega)\} - [b] \{u(\omega)\} \right] \tag{4.6}$$

and can be used directly of test/analysis correlation and/or finite element updating.

Shape correlation tools provided by ii_mac can also be used to compare frequency responses. Thus the MAC applied to FRFs is sometimes called FRAC.

4.3 Expansion methods

Expansion methods seek to estimate the motion at all DOFs of a finite element model based on measured information (typically modeshapes or frequency response functions) and prior, but not necessarily accurate, information about the structure under test in the form of a reference finite element model. As for all estimation techniques, the quality of expansion results is deteriorated by poor test results and/or poor modeling, but good results can be obtained when one or both are accurate.

The gartco demonstration illustrates modeshape expansion in the *SDT*. This section summarizes the theory and you are encouraged to download [20][18] from sdtools.com if you want more details.

4.3.1 Underlying theory for expansion methods

The unified perspective driving the *SDT* architecture is detailed in [20][18]. The proposed classification is based on how various methods combine information about

test and **modeling** errors.

Test results y_{Test} and expanded shapes q_{ex} are related by the observation equation (4.1). Test error is thus measured by a norm of the difference between the test quantity and the observed expanded shape

$$\epsilon = \left\| \{y_{Test}\} - [c]\,\{q_{ex}\} \right\|_Q^2 \tag{4.7}$$

where the choice of the Q norm is an important issue. While the Euclidian norm ($Q = I$) is used in general, a norm that takes into account an estimated variance of the various components of y_{Test} seems most appropriate. Various energy based metrics have also been considered in [21] although the motivation for using a energy norm on test results is unclear.

The expanded vector is also supposed to verify an equilibrium condition that depends on its nature. Since the model and test results don't match exactly one does not expect the expanded vector to verify this equation exactly which leads to the definition of a residual. Standard residuals are $R_j = Z(\omega_j)\phi_j$ for modeshapes and $R_j = Z(\omega)q - F$ for frequency response to the harmonic load F.

Dynamic residuals correspond to generalized loads, so they should be associated to displacement residuals and an energy norm. A standard solution [22] is to compute the static response to the residual and use the associated strain energy, which is a good indicator of modeling error,

$$\|R_j(q_{ex})\|_K^2 = \{R_j\}^T \left[\hat{K}\right]^{-1} \{R_j\} \tag{4.8}$$

where $\hat{K}$ is the stiffness of a reference FEM model and can be a mass-shifted stiffness in the presence of rigid body modes (see section 6.1.4). Variants of this energy norm of the dynamic residual can be found in [21].

Like all estimation techniques, expansion methods should clearly indicate a trade-off between test and modeling errors, since both test and model are subject to error. But modeling errors are not easily taken into account. Common expansion techniques thus only use the model to build a subspace of likely displacements.

Interpolation methods, the simplest form of subspace method are discussed in section 4.3.2. Standard subspace methods and their implementation are discussed in section section 4.3.3. Methods taking modeling errors into account are discussed in section 4.3.4.

4.3.2 Basic interpolation methods for unmeasured DOFs

Translations are always measured in a single direction. By summing the measurements of all sensors at a single physical node, it is possible for triaxial measurements to determine the 3-D motion. Using only triaxial measurements is often economically/technically impossible and is not particularly desirable. Assuming that all unmeasured motions are zero is however often not acceptable either (often distorts the perception of test modeshapes in 3-D wire frame displays).

Historically, the first solutions to this problem used geometrical interpolation methods estimating the motion in less important directions based on measurements at a few selected nodes.

Wire-frame displays can be considered as trivial interpolation methods since the motion between two test nodes is interpolated using linear shape functions.

In the *SDT*, you can easily implement interpolation methods using matrices which give the relation between measured DOFs `tdof` and a larger set of deformation DOFs `ndof`. For example, if node 2 is placed at a quarter of the distance between nodes 1 and 3. A linear interpolation for translations in the x direction would give

```
tdof = [1.01;3.01];              % List of test DOFs
exp.DOF = [1.01;2.01;3.01];      % List of DOFs to be animated
exp.def =    [1 0;3/4 1/4;0 1]; % Deformations associated
                                 % with test DOFs
```

Once the interpolation matrix (columns of `exp.def` which give the deformation at all desired DOFs `exp.DOF` for unit displacements at the sensors `tdof`) built you can animate the interpolated modeshapes using `cf.def={def,exp}`. Automated building of interpolations is supported with the `fe_sens WireExp` command which gives an example application.

For multiple interpolations, you need to include all DOFs to be animated in `exp.DOF` and include as many columns as test sensors in `exp.def`. You can check the validity of each interpolation by displaying the expansion basis `cf.def=exp`. Each deformation shown then shows how a given sensor affects animated degree of freedom `exp.DOF`.

You could also use a basis of vectors `exp.def` with non unit displacements at the measurement DOFs. The deformations at DOFs `exp.DOF` would then be found using `fe_exp(def,fe_c(exp.DOF,tdof)*exp.def,exp.def)` (which minimizes the norm of the test error (4.7) for a response within the subspace spanned by `exp.def`). The same result can be obtained by building a basis with vectors associated with unit observations

```
exp_unit=exp;
exp_unit.def=exp.def*pinv(fe_c(exp.DOF,tdof)*exp.def);
```

Spline interpolations are a way to extend a geometrical construction of the subspace but they are ill suited for complex geometries. Building subspaces using a finite element model, as discussed in the next section, is much easier. If you don't have a model, consider that segments of the wire-frame display used to visualize the test are beams with arbitrary properties and use that as a finite element model (it often gives very good results).

4.3.3 Subspace based expansion methods

If one can justify that true motion can be well represented by a vector within the subspace characterized by a basis T with no more columns than there are sensors (one assumes that the true displacement is of the form $\{q_{Ex}\} = [T]\{q_R\}$), an estimate of the true response simply obtained by minimizing test error, that is solving the least-squares problem

$$\{q_R\} = \arg\min ||\{y_{Test}\} - [c][T]\{q_R\}||_2^2 \tag{4.9}$$

Modeshape expansion based on the subspace of low frequency modes is known as **modal** [23] or **SEREP** [24] expansion. The subtle difference between the two approaches is the fact that, in the original paper, modal expansion preserved test results on test DOFs (DOFs and sensors were assumed to coincide) and interpolated motion on other DOFs. The *SDT* supports modal expansion using

```
yExp = fe_exp(yTest,sens,T)
```

where `yTest` are the measured vectors, `sens` is the sensor configuration (see `fe_sens`) or an observation matrix `c`, and `T` is a set of target modes (computed using `fe_eig` or imported from an other FE code).

An advantage of the modal methods is the fact that you can select less target modes that you have sensors which induces a smoothing of the results which can alleviate some of the problems linked to measurement/identification errors.

The study presented in [20] concludes that modal based methods perform very well when an appropriate set of target modes is selected. The only but essential limitation seems to be the absence of design/verification methodologies for target mode selection. Furthermore it is unclear whether a good selection always exists.

Modeshape expansion based on the subspace of static responses to unit displacements at sensors is known as **static** expansion or Guyan reduction [25].

When expanding modeshapes or FRFs, each deformation is associated to a frequency. It thus seems reasonable to replace the static responses by dynamic responses to loads/displacements at that frequency. This leads to dynamic expansion [26]. In general, computing a subspace for each modeshape frequency is too costly. The alternative of using a single "representative" frequency for all modes was proposed in [27] but suffers from the same limitations as choosing this frequency to be zero (Guyan reduction).

The *SDT* supports full order static and dynamic expansion using

```
yExp=fe_exp(yTest,fTest,sens,m,k,mdof)
```

where `fTest` can a single frequency (`0` for static) or have a value for each shape. In the later case, computational times are usually prohibitive so that reduced basis solutions discussed below should be used.

For tests described by observation matrices, the unit displacement problem defining static modes can be replaced by a unit load problem $[T] = [K]^{-1} [c]^T$. For structures without rigid body modes this generates the same subspace as the unit displacement problem. In other cases $[K]$ is singular and can be simply mass-shifted (replaced by $K + \alpha M$ with α usually taken small when compared to the square of the first flexible frequency, see section 6.1.4).

In practice, static expansion can be restated in the form (4.9) where T corresponds to constraint or modes associated to the load collocated to the output shape matrix characterizing sensors (see chapter 6). Restating the problem in terms of minimization is helpful if you want to compute your static responses outside the *SDT* (you won't need to import your mass and stiffness matrices but only the considered static responses).

The weakness of static expansion is the existence of a frequency limit found by computing modes of the structure with all sensors fixed. In many practical applications, this frequency limit is not that low (typically because of lack of sensors in certain areas/directions). You can easily compute this frequency limit using `fe_exp`.

Full order dynamic expansion is typically too expensive to be considered for a full order model. The *SDT* supports reduced basis dynamic expansion where you compute dynamic expansion on a subspace combining modes and static responses to loads at sensors. A typical calling sequence combining modeshape computations and static correction would be

```
[md0,f0,kd] = fe_eig(m,k,[105 30 1e2]);
T = [kd \ ((sens.ctn*sens.cna)') md0];
mdex = fe_exp(IIres.',IIpo(:,1)*2*pi,sens,m,k,mdof,T);
```

You should note however that the minimum dynamic residual expansion (MDRE) discussed in the next section typically gives better results at a marginal computational cost increase, so that you should only use dynamic expansion to expands FRFs (MDRE for FRFs is not currently implemented in `fe_exp`) or operational deflection shapes (for which modeling error is hard to define).

4.3.4 Model based expansion methods

Given metrics on test (4.7) and modeling (4.8) error, one uses a weighted sum of the two types of errors to introduce a generalized least-squares problem

$$min_{q_{j,ex}} \|R(q_{j,ex})\|_K^2 + \gamma_j \epsilon_j \tag{4.10}$$

MDRE (Minimum Dynamic Residual Expansion) assumes test errors to be zero. MDRE-WE (MDRE With test Error) sets the relative weighting (γ_j coefficient) iteratively until the desired bound on test error is reached (this is really a way to solve the least-squares problem with a quadratic inequality as proposed in [28]).

These methods are currently only implemented for modeshape expansion. When they can be used, they are really superior to subspace methods. The proper strategy to choose the error bound in MDRE-WE is still an open issue but it directly relates to the confidence you have in your model and test results.

4.4 Structural dynamic modification

While test results are typically used for test/analysis correlation and update, experimental data has direct uses. In particular,

- experimental damping ratios are often used for finite element model predictions;
- identified models can be used to predict the response after a modification (if this modification is mechanical, one talks about *structural modification*, if it is a controller one does *closed loop response* prediction);
- identified models can be used to generate control laws in active control applications;

- if some input locations of interest for structural modification have only been tested as output locations, the reciprocity assumption (see section 3.4.2) can be used to predict unmeasured transfers. But these predictions lack residual terms (see section 6.1.3) which are often important in coupled predictions.

Structural modification and closed loop predictions are important application areas of *SDT*. For closed loop predictions, users typically build state-space models with `res2ss` and then use control related tools (*Control Toolbox*, SIMULINK). If mechanical modifications can be modeled with a mass/damping/stiffness model directly connected to measured inputs/outputs, predicting the effect of a modification takes the same route as illustrated below. Mass effects correspond to acceleration feedback, damping to velocity feedback, and stiffness to displacement feedback.

The following illustrates on a real experimental dataset the prediction of a 300 g mass loading effect at a locations $1012 - z$ and $1112 - z$ (when only $1012 - z$ is excited in the `gartid` dataset used below).

```
demosdt('demo gartid est')
XF(5).idopt.reci='1 FRF';
XF(5).res=-XF(5).res; % driving 1012-z to 1012z

ind=fe_c(XF(5).dof(:,1),[1012;1112],'ind');
po_ol=IIpo;

% Using normal modes
NOR = res2nor(XF(5)); NOR.pb=NOR.cp'; S=nor2ss(NOR,'hz');
a_cl = S.a - S.b(:,ind)*S.c(ind,:)*S.a*S.a*.3;
po_cln=ii_pof(eig(a_cl)/2/pi,3,2)

% Using complex modes
S = res2ss(XF(5),'AllIO');
a_cl = S.a - S.b(:,ind)*S.c(ind,:)*S.a*S.a*.3;
po_clx=ii_pof(eig(a_cl)/2/pi,3,2)

% Frequencies
figure;subplot(211);
bar([ po_clx(:,1) po_cln(:,1)]./po_ol(:,[1 1]))
ylabel('\Delta F / F');legend('Complex modes','Normal modes')
set(gca,'ylim',[.5 1])

% Damping
```

```
subplot(212);bar([ po_clx(:,2) po_cln(:,2)]./po_ol(:,[2 2]))
ylabel('\Delta \zeta / \zeta');legend('Complex modes','Normal modes')
set(gca,'ylim',[.5 1.5])
```

Notice that the change in the sign of `XF(5).res` needed to have a positive driving point FRFs (this is assumed by `id_rm`). Reciprocity was either applied using complex (the `'AllIO'` command in `res2ss` returns all input/output pairs assuming reciprocity) or normal modes with `NOR.pb=NOR.cp'`.

Closed loop frequency predictions agree very well using complex or normal modes (as well as with FEM predictions) but damping variation estimates are not very good with the complex mode state-space model.

There is much more to *structural dynamic modification* than a generalization of this example to arbitrary point mass, stiffness and damping connections. And you can read [29] or get in touch with SDTools for our latest advances on the subject.

FEM tutorial

This chapter introduces notions needed to use finite element modeling in the *SDT*. It illustrates

- how to use model data structures,
- how to define a case (i.e. DOFs, boundary conditions, loads, ...),
- how to compute the response to a specified case,
- how to post-process results.

5.1 `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 7.6)

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

Section 5.1.1 addresses the use of the model properties GUI.

The following sections illustrate : low level input of nodes and elements in section 5.1.2; structured meshing and mesh manipulation with the `femesh` pre-processor in section 5.1.3; import of FEM models in section 5.5.1. Assembly and response computations are addressed in section 5.2.

5.1.1 GUI Access to FEM models

Graphical editing of model properties is supported by `feplot` and associated commands. Once a model is defined (see the following sections), you can display it with `feplot`. The model data structure can be manipulated graphically using the model properties GUI which can be opened using the `feplot Edit:Model Properties` menu or from the command line with `fecom('pro modelinit')`.

For example

```
model=femesh('test ubeam plot');
fecom('promodelinit');
```

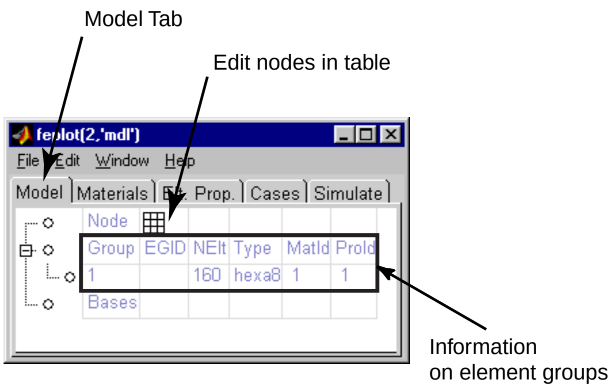


Figure 5.1: Model properties interface.

The model properties figure contains the following tabs

`model`	Node editing and element group display
`Materials`	Material editing, see section 5.1.4
`Properties`	Element property (for bar, beam and shells elements) editing, see section 5.1.4
`Case`	Loads and boundary conditions editing, see section 5.2
`Simulate`	Static, modal and transient response editing, see section 5.3

While GUI access may be useful in a learning phase, script access (through command line or `.m` files) is important. Variable handles let you modify the model properties contained in the model properties GUI. For the model contained in an `feplot` figure, you obtain a SDT Handle to the `feplot` figure with `cf=feplot` and a variable handle to the model data structure with `cf.mdl`.

5.1.2 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.

The geometry is declared in the `model.Node` matrix (see section 7.1). 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
```

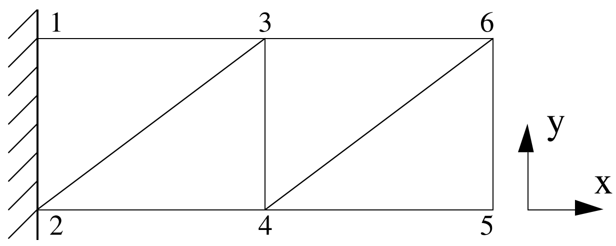


Figure 5.2: FE model.

```
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 7.1) 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

```
cf=feplot; cf.model=model;       % create feplot axes
fecom(';view2;textnode;triax;'); % manipulate axes
```

The demo_fe script illustrates uses of this model.

5.1.3 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
cf=feplot; cf.model=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('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
cf=feplot; cf.model=model;
fecom(';triaxon;view3;view y+180;view s-10');
```

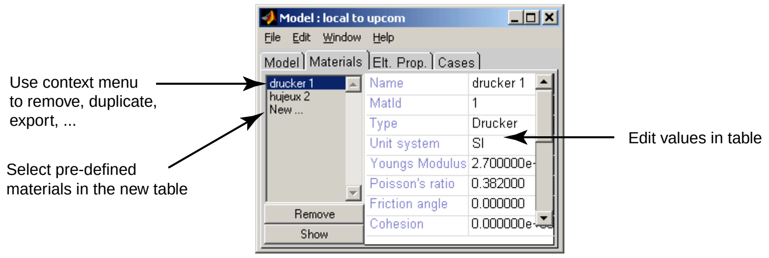
`femesh` allows many other manipulations (translation, rotation, symmetry, extrusion, generation by revolution, refinement by division of elements, selection of groups, nodes, elements, edges, etc.) which are detailed in the *Reference* section.

Other more complex examples are treated in the following demonstration scripts `d_plate`, `beambar`, `d_ubeam`, `gartfe`.

5.1.4 Handling material and element properties

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

You can edit material properties using the `Materials` tab of the `Model Properties` figure which lists current materials and lets you choose new ones from the database of each material type. `m_elastic` is the only material function defined for the base *SDT*. It supports elastic materials and linear acoustic fluids.



drucker 1
hujeux 2
New ...

Name	drucker 1
MatId	1
Type	Drucker
Unit system	SI
Youngs Modulus	2.700000e-
Poisson's ratio	0.382000
Friction angle	0.000000
Cohesion	0.000000e-

Figure 5.3: Property tab.

Similarly the `Property` tab lets you edit element properties. `p_beam` `p_shell` and `p_spring` are supported element property functions.

The properties are stored with one property per row in pl (see section 7.3) and il (see section 7.4) model fields.

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

```
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 ; ... % longerc
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).

5.1.5 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.

Use of local coordinate systems is illustrated in section 4.1.1 where a local basis is defined for test results.

`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 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 5.2.2) 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.

5.2.1 Cases GUI

Graphical editing of case properties is supported by the case tab of the model properties GUI (see section 5.1.1).

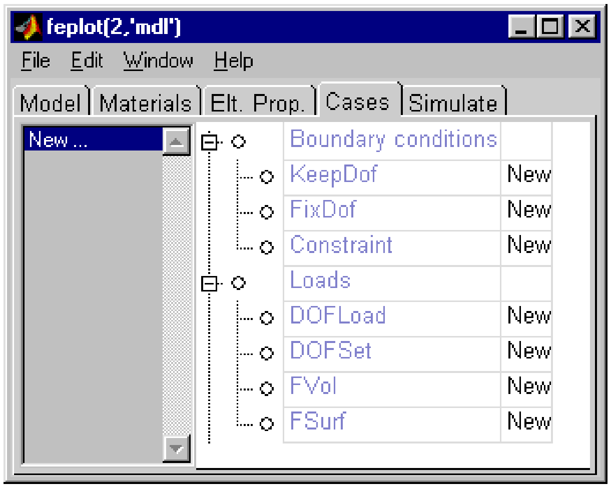


Figure 5.4: Cases properties tab.

When selecting `New ...` in the case property list, as shown in the figure, you get a list of currently supported case properties. You can add a new property by clicking on the associated `new` cell in the table. Once a property is opened you can typically edit it graphically. The following sections show you how to edit these properties trough command line or `.m` files.

5.2.2 Boundary conditions and constraints

Boundary conditions and constraints are described in `Case.Stack` using `KeepDof`, `FixDof` and `Rigid` case entries (see section 5.2).

`KeepDof` and `FixDof` entries are used to easily impose zero displacement on some DOFs. To treat the two bay truss example of section 5.1.2, one will for example use

```
model=femesh('test 2bay plot');
model=fe_case(model,'SetCase1', ...          % defines a new case
  'KeepDof','2-D motion',[.01 .02 .06]', ...
  'FixDof','Clamp edge',[1 2]');
fecom('promodelinit') % open model GUI
```

When assembling the model with the specified `Case` (see section 5.2), 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. In the two bay truss case,

```
model=femesh('test 2bay plot');
mdof = feutil('getdof group1:2',model);
adof = fe_c(mdof,[.01 .02 .06]','dof'); % 2-D motion
i1   = femesh('findnode x==0');         % left edge nodes
adof = fe_c(adof,i1 ,'dof',2);          % clamp x==0 plane
model=fe_case(model,'SetCase1', ...     % defines a new case
    'KeepDof','final DOF list',adof);
fecom('promodelinit') % open model GUI
```

finds all DOFs in element groups 1 and 2 of FEelt, 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 7.13.

5.2.3 Loads

Loads are described in Case.Stack using DOFLoad, FVol and FSurf case entries (see fe_case and section 7.7).

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

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

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

```
model = femesh('testubeam plot');
data=struct('sel','x==-.5', ...
   'eltsel','withnode {z>1.25}','def',1,'DOF',.19);
Case1=struct('Stack',{{'Fsurf','Surface load',data}});
Load = fe_load(model,Case1); cf.def=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

```
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,'AddToCase 1','DOFLoad','Point load 1',data);
data  = struct('DOF',365.03,'def',1);
model = fe_case(model,'AddToCase 1','DOFLoad','Point load 2',data);
Load  = fe_load(model,'Case1');
cf.def= 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');
```

5.2.4 Sensors

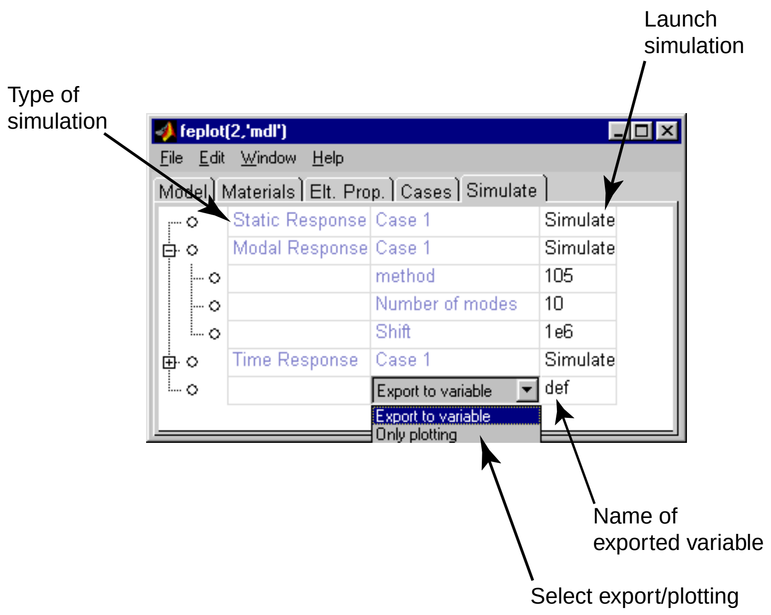
For simulations, you may only want to measure partial information about the model. This is supported by the use of SensDOF entries associated to a case. The following example defines three sensors and builds the associated state-space model

```
model=demosdt('demo ubeam mix');
model=fe_case(model,'addtocase', ...
  'SensDof','Outputs',[343.01 343.02 347.03]')
[sys,T] = fe2ss('free 6 10',model);
figure(1);qbode(sys,linspace(100,1e3,1024)'*2*pi,'plot');
```

5.3 Computing the response

5.3.1 Simulate GUI

Access to standard solvers is provided through the Simulate tab of the Model properties figure. Experienced users will typically use the command line equivalent to these tabs as detailed in the following sections.



Static Response	Case 1	Simulate
Modal Response	Case 1	Simulate
	method	105
	Number of modes	10
	Shift	1e6
Time Response	Case 1	Simulate
	Export to variable	def

Figure 5.5: Simulation properties tab.

5.3.2 Static responses

The computation of the response to static loads is a typical problem. Once loads and boundary conditions are defined in a case as shown in previous sections, the static response may be computed using the fe_simul function.

This is an example of the 3D beam subjected to various type of loads (points, surface and volume loads) and clamped at its base :

```
model=demosdt('demo ubeam vol'); % Initialize a test
def=fe_simul('static',model,'Case 1');% Compute static response
cf=feplot; cf.def=def;% post-process
cf.sel={'Groupall','ColorDataStressMises'}
```

Low level calls may also be used. For this purpose it is generally simpler to create system matrices that incorporate the boundary conditions.

fe_c (for point loads) and fe_load (for distributed loads) can then be used to define unit loads (input shape matrix using *SDT* terminology). For example, a unit vertical input (DOF .02) on node 6 can be simply created by

```
model=demosdt('demo2bay');  Case=fe_case(model,'gett'); %init
% Compute point load
b = fe_c(Case.DOF,[6.02],1)';
```

In many cases the static response can be computed using `Static=kr\b`. For very large models, you will prefer

```
 kd=ofact(k); Static = kd\b; ofact('clear',kd);
```

For repeated solutions with the same factored stiffness, you should build the factored stiffness `kd=ofact(k)` and then `Static = kd\b` as many times are needed. Note that `fe_eig` can return the stiffness that was used when computing modes (when using methods without DOF renumbering).

For models with rigid body modes or DOFs with no stiffness contribution (this happens when setting certain element properties to zero), the user interface function `fe_reduc` gives you the appropriate result in a more robust and yet computationally efficient manner

```
Static = fe_reduc('flex',m,k,mdof,b);
```

5.3.3 Normal modes (partial eigenvalue solution)

`fe_eig` computes mass normalized normal modes. The simple call `def=fe_eig(model)` should only be used for very small models (below 100 DOF). In other cases you will typically only want a partial solution. A typical call would have the form

```
model = demosdt('demo ubeam plot');
cf.def=fe_eig(model,[6 12 0]);  % 12 modes with method 6
fecom('colordata stress mises')
```

You should read the `fe_eig` reference section to understand the qualities and limitations of the various algorithms for partial eigenvalue solutions.

You can also load normal modes computed using a finite element package (see section 5.5.1). If the finite element package does not provide mass normalized modes, but a diagonal matrix of generalized masses `mu` (also called modal masses). Mass normalized modeshapes will be obtained using

```
ModeNorm = ModeIn * diag( diag(mu).^(-1/2) );
```

If a mass matrix is given, an alternative is to use `mode = fe_norm(mode,m)`. When both mass and stiffness are given, a Ritz analysis for the complete problem is obtained using `[mode,freq] = fe_norm(mode,m,k)`.

Note that loading modes with in ASCII format 8 digits is usually sufficient for good accuracy whereas the same precision is very often insufficient for model matrices (particularly the stiffness).

5.3.4 State space and other modal models

A typical application of *SDT* is the creation of input/output models in the normal mode `nor`, state space `ss` or FRF `xf` form. (The *SDT* does not replicate existing functions for time response generation such as `lsim` of the *Control Toolbox* which creates time responses using a model in the state-space form).

The creation of such models combines two steps creation of a modal or enriched modal basis; building of input/output model given a set of inputs and outputs.

As detailed in section 5.3.3 a modal basis can be obtained with `fe_eig` or loaded from an external FEM package. Inputs and outputs are easily handled using case entries corresponding to loads (`DofLoad`, `DOFSet`, `FVol`, `FSurf`) and sensors (`SensDOF`).

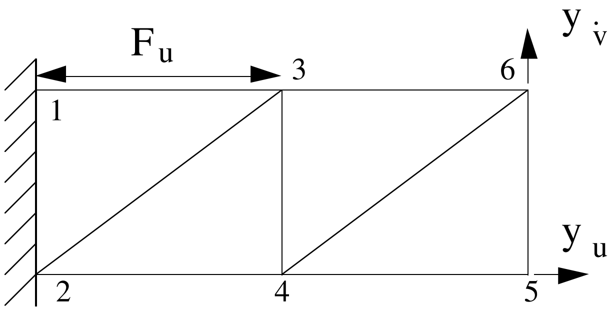


Figure 5.6: Truss example.

For the two bay truss example shown above, the following script defines a load as the relative force between nodes 1 and 3, and translation sensors at nodes 5 and 6

```
model=demosdt('demo2bay');
DEF=fe_eig(model,[2 5]);  % compute 5 modes

% Define loads and sensors
Load=struct('DOF',[3.01;1.01],'def',[1;-1]);
Case=fe_case('DofLoad','Relative load',Load, ...
             'SensDof','Tip sensors',[5.01;6.02]);
```

```
% Compute FRF and display
w=linspace(80,240,200)';
figure(1);clf;xf=nor2xf(DEF,.01,Case,w,'hz plot');
```

You can easily obtain velocity or acceleration responses using

```
xf=nor2xf(DEF,.01,Case,w,'hz vel plot');
xf=nor2xf(DEF,.01,Case,w,'hz acc plot');
```

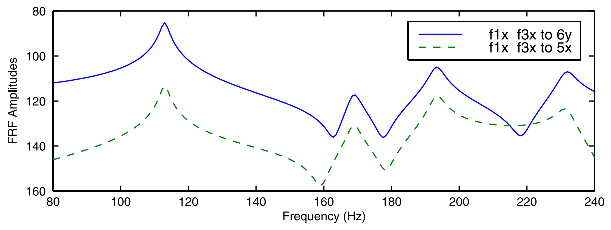


Figure 5.7: FRF synthesis : with and without static correction.

As detailed in section 6.1.3, it is desirable to introduce a static correction for each input. `fe2ss` builds on `fe_reduc` to provide optimized solutions where you compute both modes and static corrections in a single call and return a state-space (or normal mode model) and associated reduction basis. Thus

```
model=demosdt('demo2bay');
DEF=fe_eig(model,[2 5]);  % compute 5 modes
Load=struct('DOF',[3.01;1.01],'def',[1;-1]);
Case=fe_case(model,'AddToCase1','DofLoad','Relative load',Load, ...
             'SensDof','Tip sensors',[5.01;6.02]);
[SYS,DEF] = fe2ss('free 2 3',model);
w=linspace(80,240,200);
figure(1);clf;xf=nor2xf(DEF,.01,Case,w,'hz plot');
```

computes 3 modes using a full solution (*Eigopt*=`[2 3 0]`), appends the static response to the input shape matrix `b`, and builds the state-space model corresponding to modal truncation with static correction (see section 6.1.3). **Note** that the load and sensor definitions where now added to the case in `model` since that case also contains boundary condition definitions which are needed in `fe2ss`.

The different functions using normal mode models support further model truncation. For example, to create a model retaining the first four modes, one can use

```
model=demosdt('demo2bay');
DEF=fe_eig(model,[2 12]);  % compute 12 modes
Case=fe_case('DofLoad','Horizontal load',3.01, ...
             'SensDof','Tip sensors',[5.01;6.02]);
SYS =nor2ss(DEF,.01,Case,1:4);
ii_pof(eig(SYS.a)/2/pi,3)  % Frequency (Hz), damping
```

A static correction for the displacement contribution of truncated modes is automatically introduced in the form of a non-zero `d` term. When considering velocity outputs, the accuracy of this model can be improved using static correction modes instead of the `d` term. Static correction modes are added if a roll-off frequency `fc` is specified (this frequency should be a decade above the last retained mode and can be replaced by a set of frequencies)

```
SYS =nor2ss(DEF,.01,Case,1:4,[2e3 .2]);
ii_pof(eig(SYS.a)/2/pi,3,1)  % Frequency (Hz), damping
```

Note that `nor2xf` always introduces a static correction for both displacement and velocity.

For damping, you can use uniform modal damping (a single damping ration `damp=.01` for example), non uniform modal damping (a damping ratio vector `damp`), non-proportional modal damping (square matrix `ga`), or hysteretic (complex `DEF.data`). This is illustrated in `demo_fe`.

5.3.5 Manipulating large finite element models

The flexibility given by the Matlab language comes at a price for large finite element computations. The two main bottlenecks are model assembly and static computations.

During assembly compiled elements provided with OpenFEM allow much faster element matrix evaluations (since these steps are loop intensive they are hard to optimize in Matlab). The `sp_util.mex` function alleviates element matrix assembly and large matrix manipulation problems (at the cost of doing some very dirty tricks like modifying input arguments).

For static computations, the `ofact` object allows method selection. Currently the most efficient (and default `ofact` method) is the multi-frontal sparse solver `spfmex`. This solver automatically performs equation reordering so this needs not be done elsewhere. It does not use the Matlab memory stack which is more efficient for large problems but requires `ofact('clear')` calls to free memory associated with a given factor.

With other static solvers (MATLAB `lu` or `chol`, or *SDT* true skyline `sp_util` method) you need to pay attention to equation renumbering. When assembling large models, `fe_mk` will automatically renumber DOFs to minimize matrix bandwidth (for partial backward compatibility automatic renumbering is only done above 1000 DOF).

The real limitation on size is linked to performance drops when swapping. If the factored matrix size exceeds physical memory available to MATLAB in your computer, performance tends to decrease drastically. The model size at which this limit is found is very much model/computer dependent.

Finally in `fe_eig`, method 6 (IRA/Sorensen) uses low level BLAS code and thus tends to have the best memory performance for eigenvalue computations.

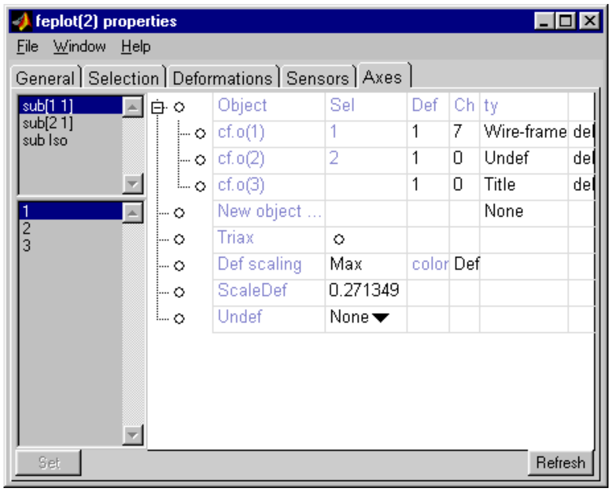
Note finally, that you may want to run MATLAB with the `-nojvm` option turned on since it increases the memory addressable by MATLAB.

For out-of-core operations (supported by `fe_mk`, `upcom`, `nasread` and other functions). SDT creates temporary files whose names are generated `tempname`.You may need to redefine your own `tempdir.m` function and make sure that it is properly placed in the MATLAB path using `which('tempdir','-all')`. If your own `tempdir` is not shown first in the list use `addpath ... -begin` commands to put its directory first in your path.

5.4 Post-processing with `feplot`

`feplot` supports a number of display types for FE results. 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 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).

Most demonstrations linked to finite element modeling (see section 1.1 for a list) give examples of how to use `feplot` and `fecom`. To get you started, you can try the following which gives you a rapid overview of the capabilities of the `feplot` finite element visualization interface.



sub[1 1]
sub[2 1]
sub Iso

1
2
3

Object	Sel	Def	Ch	ty	
cf.o(1)	1	1	7	Wire-frame	de
cf.o(2)	2	1	0	Undef	de
cf.o(3)		1	0	Title	de
New object ...				None	
Triax	o				
Def scaling	Max	color	Def		
ScaleDef	0.271349				
Undef	None ▼				

Figure 5.8: feplot interface.

5.4.1 Starting the visualization interface

Load the data from the `gartfe` demo, get `cf` a *SDT* `handle` for a `feplot` figure, set the model for this figure and get the standard 3D view of the structure

```
load sdt_gart FEnode FEelt mdof md0 f0
cf=feplot;  cf.model={FEnode,FEelt};
fecom('view3');
```

Note that `cf.model=UFS(1)` or `cf.model=Up` would be acceptable for database wrapper or type 3 superelement input.

5.4.2 Using `iimouse` commands

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`, `3`, `2` keys to rotate the plot (look at the `iimouse` help for more possibilities).

open the `contextmenu` associated with your plot using the right mouse button and select `cursor`. See how the cursor allows you to know node numbers and positions. Use the left mouse button to get more info on the current node (when you have more than one object, the `n` key is used to go to the next object). Use the right button to exit the `cursor` mode.

Notice the other things you can do with the `ContextMenu` : open the `feplot` and model properties figure, display the orientation triax, show or don't show the undeformed structure, display node numbers (note that for large models this is very long), set standard views and view defaults, ...

5.4.3 Viewing deformations

The following initializes a set of deformations, shows deformation 7 (first flexible mode), opens the `feplot` properties figure and shows the `Deformations` tab.

```
load sdt_gart; cf=feplot;  cf.model={FEnode,FEelt};
cf.def(1)={md0,mdof,f0};  fecom('ch7');
fecom('pro');fecom('curtab def')
```

Note that `cf.def=XF(5)` or `cf.def=Up.def` would be acceptable for database wrapper (`XF(5)` is used by `idcom` to store identification results) or type 3 superelement input.

Scan through the various deformations using the `+/-` buttons/keys or cliking in the deformations list in the `Deformations` tab. From the command line you can use `fecom ch` commands.

Animate the deformations by clicking on the button. Notice how you can still change the current deformation, rotate, etc. while running the animation. Animation properties can be modified with `fecom Anim` commands or in the `General` tab of the `feplot properties` figure.

Modeshape scaling can be modified with the `l/L` key, with `fecom scale` commands or in the `Axes` tab of the `feplot properties` figure.

You may also want to visualize the measurement at various sensors using a stick or arrow sensor visualization (`fecom showsens` or `showarrow`). On such plots, you can label some or all degrees of freedom using `fecom ('doftext',idof)`.

```
load sdt_gart; cf=feplot; cf.model={FEnode,FEelt};
cf.def(1)={IIres,sdof,IIpo};
cf.sens=sdof;
fecom(';showarrow;textdof;ch1;scd.3');
```

Look at the `fecom` reference section to see what modifications of displayed plots are available.

5.4.4 Superposing deformations

Modeshape superposition is an important application (see plot of section 3.1.1) which is supported by initializing deformations with the two deformation sets given sequentially and a fecom ch command declaring more than one deformation. For example you could compare two sets of deformations using

```
load sdt_gart; cf=feplot;  cf.model={FEnode,FEelt};
cf.def(1)={md0,mdof,f0};
cf.def(2)={md0+rand(size(md0))/5,mdof,f0};
fecom('show2def');
fecom('scalematch');
```

where the scalematch command is used to compare deformations with unequal scaling. You could also show two deformations in the same set

```
load sdt_gart; cf=feplot;  cf.model={FEnode,FEelt};
cf.def(1)={md0,mdof,f0};
fecom(';showline; ch7 10')
```

The -,+ buttons/commands will then increment both deformations numbers (overlay 8 and 11, etc.).

5.4.5 Element selections

Element selections play a central role in feplot. They allow selection of a model subpart (see section 7.12) and contain color information. The following example selects some groups and defines color to be the z component of displacement or all groups with strain energy deformation (see fecom ColorData commands)

```
cf=demosdt('demo gartfe plot');
cf.sel(1)={'group4:9 & group ~=8','colordata z'};
pause
cf.def=fe_eig(cf.mdl,[6 20 1e3]);
cf.sel(1)={'group all','colordata enerk'};
fecom('colorbar');
```

You can also have different objects point to different selections. This model has an experimental mesh stored in element group 11 (it has EGID -1). The following commands define a selection for the FEM model (groups 1 to 10) and one for the test wire frame (it has EGID<0). The first object cf.o(1) displays selection 1 as a surface plot (ty1 with a blue edge color. The second object displays selection to with a thick red line.

```
load sdt_gart; cf=feplot;  cf.model={FEnode,FEelt};
cf.sel(1)={'group1:10'};  cf.sel(2)='egid<0';
cf.o(1)={'ty1 def1 sel1','edgecolor','b'}
cf.o(2)={'ty2sel2','edgecolor','r','linewidth',2}
```

Note that you can use node selection commands to display some node numbers. For example try `fecom('textnode egid<0 & y>0')`.

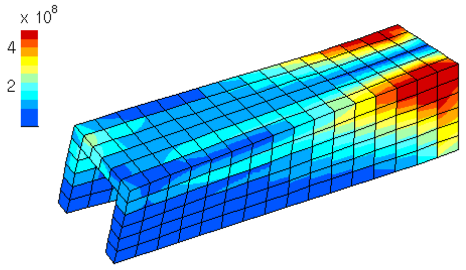


Figure 5.9: Strain energy deformation.

5.4.6 Other information

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`

Use the `Feplot:Sub commands:Sub IsoViews` (same as `iicom('subiso')`) to get a plot with four views of the same mode. Use `iicom('sub2 2 step')` to get four views of different modes.

Use the `Feplot:Show` menu to generate standard plots. `Feplot:Misc` for `Triax`, `Undef`, channel selection ...

5.5 Interfacing with other FEM codes

The base *SDT* supports reading/writting of test related Universal files and basic NASTRAN bulk files. All other interfaces are packaged in the FEMLink extension.

FEMLink is installed within the base SDT but can only be accessed by licensed users.

5.5.1 Importing models from other codes

Interfaces currently available are

`ans2sdt`	reads ANSYS binary files (this function is part of FEMLink)
`nasread`	reads the MSC/NASTRAN [30] `.f06` output file (matrices, tables, real modes, displacements, applied loads, grid point stresses), input `bulk` file (nodes, elements, properties). FEMLink provides extensions of the basic `nasread`, `output2` to model format conversion including element matrix reading, `output4` file reading, advanced bulk reading capabilities).
`naswrite`	writes formatted input to the `bulk data deck` of MSC/NASTRAN (part of SDT), FEMLink adds support for case writing.
`readnopo`	This OpenFEM function reads MODULEF models in binary format.
`perm2sdt`	reads PERMAS ascii files (this function is part of FEMLink)
`ufread`	reads results in the Universal File format (in particular, types: 55 analysis data at nodes, 58 data at DOF, 15 grid point, 82 trace line). Reading of additional FEM related file types is supported by FEMLink through the `uf_link` function.
`ufwrite`	writes results in the Universal File format. *SDT* supports writing of test related datasets. FEMLink supports FEM model writing.

You will find an up to date list of interfaces with other FEM codes at www.sdtools.com/tofromfem.html). Import of model matrices in discussed in section 5.5.2.

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 (please send your suggestions at `suggest@sdtools.com`).

5.5.2 Importing model matrices from other codes

FEMLink handles importing element matrices for NASTRAN and ANSYS. For NASTRAN, run an eigenvalue computation with the `PARAM,POST,-4` card and load the superelement using

```
upcom('load UpcomFile');
Up=nasread(Up,'NastranOutput.op2');
```

For ANSYS, run an eigenvalue computation and use

```
Up=ans2sdt('buildup test1')
```

which reads the test1.rst and test1.emat binary files for model description and element matrices and generates the upcom superelement Up saved in file test1.mat.

For full FEM model matrices you can proceed as follows. A full FEM model matrix is most appropriately integrated as a superelement. The model would typically be composed of

- a mass m and stiffness matrix k linked to DOFs mdof which you have imported with your own code (for example, using nasread output2 or output4 and appropriate manipulations to create mdof). Note that the ofact object provides translation from skyline to sparse format.
- an equivalent mesh defined using standard *SDT* elements. This mesh will be used to plot the imported model and possibly for repeating the model in a periodic structure

A unique superelement called cell will be simply created using

```
fesuper('new cell',FEnode,FEel0)
global SEcell
SEcell.DOF=mdof;
SEcell.K={m,k}; SE.Opt=[1 0;2 1];
```

where you should note the declaration of type 1 (stiffness) and type 2 (mass) matrices for the superelement k2 and k1 matrices. In this example you will get back the element matrices using

```
model=struct('Node',FEnode,'Elt',[Inf abs('cell')]);
[m,k,mdof]=fe_mk(model);
```

Note that numerical precision is very important when importing model matrices. Storing matrices in 8 digit ASCII format is very often not sufficient.

A weighting coefficient can easily be associated to matrices of *SDT* superelements. Here this would be done by defining an element property and setting the coefficients when calling fe_mk.

```
fesuper('set cell ProID 99')
model=struct('Node',FEnode,'Elt',[Inf abs('cell')],...
             'pl',[99 CoefM CoefK]);
[m,k,mdof]=fe_mk(model);
```

For structures where the imported component model is repeated, you may want to make the superelement generic so that you can use it several times. For example, if cell contains a truss bay. You can create a 10 bay truss and compute its first 10 modes using

```
FEel0=fesuper('make cell generic');
femesh('repeatsel 10 1 0 0');
 model=struct('Node',FEnode,'Elt',FEel0,'pl',[99 1 1]);
def=fe_eig(model,[6 10 1e3]);
cf=feplot;cf.model=model;cf.def=def;
```

Superelement based substructuring is demonstrated in d_cms2 which gives you a working example where model matrices are stored in a generic superelement.

6

Advanced FEM tools

6.1 Model reduction theory

Finite element models of structures need to have many degrees of freedom to represent the geometrical detail of complex structures. For models of structural dynamics, one is however interested in

- a restricted frequency range ($s = i\omega \in [\omega_1 \ \ \omega_2]$)
- a small number of inputs and outputs (b, c)
- a limited parameter space α (updated physical parameters, design changes, non-linearities, etc.)

These restrictions on the expected predictions allow the creation of low order models that accurately represent the dynamics of the full order model in all the considered loading/parameter conditions.

Model reduction notions are key to many *SDT* functions of all areas: to motivate residual terms in pole residue models (`id_rc`, `id_nor`), to allow fine control of model order (`nor2ss`, `nor2xf`), to create normal models of structural dynamics from large order models (`fe2ss`, `fe_reduc`), for test measurement expansion to the full set of DOFs (`fe_exp`), for substructuring using superelements (`fesuper`, `fe_coor`), for parameterized problems including finite element model updating (`upcom`).

6.1.1 General framework

Model reduction procedures are discrete versions of Ritz/Galerkin analyzes: they seek solutions in the subspace generated by a reduction matrix T. Assuming $\{q\} = [T]\{q_R\}$, the second order finite element model (2.1) is projected as follows

$$\begin{gathered}\left[T^TMTs^2 + T^TCTs + T^TKT\right]_{NR\times NR}\{q_R(s)\} = \left[T^Tb\right]_{NR\times NA}\{u(s)\}_{NA\times 1}\\ \{y(s)\}_{NS\times 1} = [cT]_{NS\times NR}\{q_R(s)\}_{NR\times 1}\end{gathered} \tag{6.1}$$

Modal analysis, model reduction, component mode synthesis, and related methods all deal with an appropriate selection of singular projection bases ($[T]_{N\times NR}$ with $NR \ll N$). This section summarizes the theory behind these methods with references to other works that give more details.

The solutions provided by *SDT* make two further assumptions which are not hard limitations but allow more consistent treatments while covering all but the most exotic problems. The projection is chosen to preserve reciprocity (left multiplication by T^T and not another matrix). The projection bases are assumed to be real.

An accurate model is defined by the fact that the input/output relation is preserved for a given frequency and parameter range

$$[c]\,[Z(s,\alpha)]^{-1}\,[b] \approx [cT]\left[T^T Z(s,\alpha) T\right]^{-1}\left[T^T b\right] \tag{6.2}$$

Traditional modal analysis, combines normal modes and static responses. *Component mode synthesis* methods extend the selection of boundary conditions used to compute the normal modes. The *SDT* further extends the use of reduction bases to parameterized problems.

A key property for model reduction methods is that the input/output behavior of a model only depends on the vector space generated by the projection matrix T. Thus range(T) = range$(\tilde{T})$ implies that

$$[cT]\left[T^T Z T\right]^{-1}\left[T^T b\right] = \left[c\tilde{T}\right]\left[\tilde{T}^T Z \tilde{T}\right]^{-1}\left[\tilde{T}^T b\right] \tag{6.3}$$

This **equivalence property** is central to the flexibility provided by the *SDT* in CMS applications (it allows the decoupling of the reduction and coupled prediction phases) and modeshape expansion methods (it allows the definition of a static/dynamic expansion on sensors that do not correspond to DOFs).

6.1.2 Normal mode models

Normal modes are defined by the eigenvalue problem

$$-\,[M]\,\{\phi_j\}\,\omega_j^2 + [K]_{N\times N}\,\{\phi_j\}_{N\times 1} = \{0\}_{N\times 1} \tag{6.4}$$

based on inertia properties (represented by the positive definite mass matrix M) and underlying elastic properties (represented by a positive semi-definite stiffness K). The matrices being positive there are N independent eigenvectors $\{\phi_j\}$ (forming a matrix noted $[\phi]$) and eigenvalues ω_j^2 (forming a diagonal matrix noted $\left[\backslash\omega_{j\backslash}^2\right]$).

As solutions of the eigenvalue problem (6.4), the full set of N normal modes verify two **orthogonality conditions** with respect to the mass and the stiffness

$$[\phi]^T\,[M]\,[\phi] = \left[\backslash\mu_{j\backslash}\right]_{N\times N} \quad \text{and} \quad [\phi]^T\,[K]\,[\phi] = \left[\backslash\mu_j\omega_{j\backslash}^2\right] \tag{6.5}$$

where μ is a diagonal matrix of modal masses (which are quantities depending uniquely on the way the eigenvectors ϕ are scaled).

In the *SDT*, the normal modeshapes are assumed to be mass normalized so that $[\mu] = [I]$ (implying $[\phi]^T [M] [\phi] = [I]$ and $[\phi]^T [K] [\phi] = \left[\backslash \omega_j^2 \backslash\right]$). The **mass normalization** of modeshapes is independent from a particular choice of sensors or actuators.

Another traditional normalization is to set a particular component of $\tilde{\phi}_j$ to 1. Using an output shape matrix this is equivalent to $c_l \tilde{\phi}_j = 1$ (the observed motion at sensor c_l is unity). $\tilde{\phi}_j$, the modeshape with a component scaled to 1, is related to the mass normalized modeshape by $\tilde{\phi}_j = \phi_j / (c_l \phi_j)$.

$$m_j(c_l) = (c_l \phi_j)^{-2}$$

is called the **modal or generalized mass** at sensor c_l. A large modal mass denotes small output. For rigid body translation modes and translation sensors, the modal mass corresponds to the mass of the structure. If a diagonal matrix of generalized masses `mu` is provided and `ModeIn` is such that the output c_l is scaled to 1, the mass normalized modeshapes will be obtained by

```
ModeNorm = ModeIn * diag(diag(mu).^(-1/2));
```

Modal stiffnesses are are equal to

$$k_j(c_l) = (c_l \phi_j)^{-2} \omega_j^2$$

The use of mass-normalized modes, simplifies the normal mode form (identity mass matrix) and allows the direct comparison of the contributions of different modes at similar sensors. From the orthogonality conditions, one can show that, for an undamped model and mass normalized modes, the dynamic response is described by a sum of modal contributions

$$[\alpha(s)] = \sum_{j=1}^{N} \frac{\{c\phi_j\} \left\{\phi_j^T b\right\}}{s^2 + \omega_j^2} \tag{6.6}$$

which correspond to pairs of complex conjugate poles $\lambda_j = \pm i\omega_j$.

In practice, only the first few low frequency modes are determined, the series in (6.6) is truncated, and a correction for the truncated terms is introduced (see section 6.1.3).

Note that the concept of effective mass [31], used for rigid base excitation tests, is very similar to the notion of generalized mass.

6.1.3 Static correction to normal mode models

Normal modes are computed to obtain the spectral decomposition (6.6). In practice, one distinguishes modes that have a resonance in the model bandwidth and need to be kept and higher frequency modes for which one assumes $\omega \ll \omega_j$. This assumption leads to

$$[c]\left[Ms^2+K\right]^{-1}[b] \approx \sum_{j=1}^{N_R} \frac{[c]\{\phi_j\}\{\phi_j\}^T[b]}{s^2+\omega_j^2} + \sum_{j=N_R+1}^{N} \frac{[c]\{\phi_j\}\{\phi_j\}^T[b]}{\omega_j^2} \tag{6.7}$$

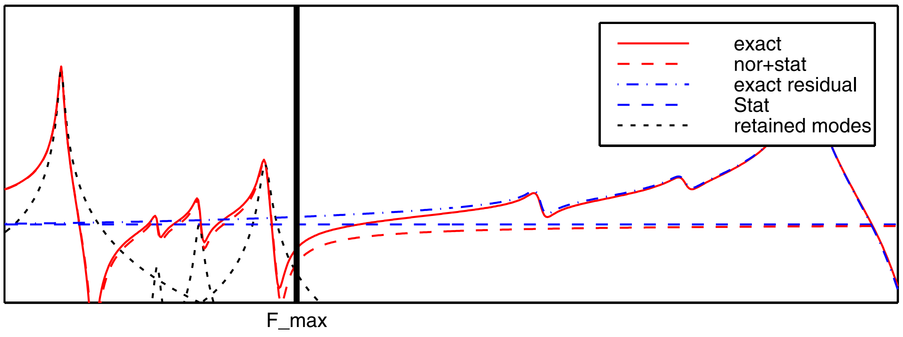


Figure 6.1: Normal mode corrections.

For the example treated in the demo_fe script, the figure shows that the exact response can be decomposed into retained modal contributions and an exact residual. In the selected frequency range, the exact residual is very well approximated by a constant often called the **static correction**.

The use of this constant is essential in identification phases and it corresponds to the E term in the pole/residue models used by id_rc (see under res page 37).

For applications in reduction of finite element models, a little more work is typically done. From the orthogonality conditions (6.5), one can easily show that for a structure with no rigid body modes (modes with $\omega_j = 0$)

$$[T_A] = [K]^{-1}[b] = \sum_{j=1}^{N} \frac{\{\phi_j\}\left\{\phi_j^T b\right\}}{\omega_j^2} \tag{6.8}$$

The static responses $K^{-1}b$ are called **attachment modes** in Component Mode Synthesis applications [32]. The inputs $[b]$ then correspond to unit loads at all interface nodes of a coupled problem.

One has historically often considered **residual attachment modes** defined by

$$[T_{AR}] = [K]^{-1}[b] - \sum_{j=1}^{NR} \frac{\{\phi_j\}\left\{\phi_j^T b\right\}}{\omega_j^2} \tag{6.9}$$

where NR is the number of normal modes retained in the reduced model.

The vector spaces spanned by $[\phi_1 \dots \phi_{NR} \ \ T_A]$ and $[\phi_1 \dots \phi_{NR} \ \ T_{AR}]$ are clearly the same, so that reduced models obtained with either are dynamically equivalent. For use in the *SDT*, you are encouraged to find a basis of the vector space that diagonalizes the mass and stiffness matrices (normal mode form which can be easily obtained with `fe_norm`).

Reduction on modeshapes is sometimes called the **mode displacement method**, while the addition of the **static correction** leads to the **mode acceleration method**.

When reducing on these bases, the selection of retained normal modes guarantees model validity over the desired frequency band, while adding the static responses guarantees validity for the spatial content of the considered inputs. The reduction is only valid for this restricted spatial/spectral content but very accurate for solicitations that verify these restrictions.

Defining the bandwidth of interest is a standard difficulty with no definite answer. The standard, but conservative, criterion (attributed to Rubin) is to keep modes with frequencies below 1.5 times the highest input frequency of interest.

6.1.4 Static correction with rigid body modes

For a system with NB rigid body modes kept in the model, $[K]$ is singular. Two methods are typically considered to overcome this limitation.

The approach traditionally found in the literature is to compute the static response of all flexible modes. For NB rigid body modes, this is given by

$$[K]^{*}[b] = \sum_{j=NB+1}^{N} \frac{\{\phi_j\}\left\{\phi_j^T b\right\}}{\omega_j^2} \tag{6.10}$$

This corresponds to the definition of **attachment modes** for free floating structures [32]. The flexible response of the structure can actually be computed as a static problem with an iso-static constraint imposed on the structure (use the `fe_reduc flex` solution and refer to [33] or [34] for more details).

The approach preferred in the *SDT* is to use a mass-shifted stiffness leading to the definition of **shifted attachment modes** as

$$[T_{AS}] = [K + \alpha M]^{-1}[b] = \sum_{j=1}^{N} \frac{\{\phi_j\}\left\{\phi_j^T b\right\}}{(\omega_j^2 + \alpha)} \tag{6.11}$$

While these responses don't exactly span the same subspace as static corrections, they can be computed using the mass-shifted stiffness used for eigenvalue computations. For small mass-shifts (a fraction of the lowest flexible frequency) and when modes are kept too, they are a very accurate replacement for attachment modes. It is the opinion of the author that the additional computational effort linked to the determination of true attachment modes is not mandated and shifted attachment modes are used in the *SDT*.

6.1.5 Other standard reduction bases

For coupled problems linked to model substructuring, it is traditional to state the problem in terms of imposed displacements rather than loads.

Assuming that the imposed displacements correspond to DOFs, one seeks solutions of problems of the form

$$\begin{bmatrix} Z_{II}(s) & Z_{IC}(s) \\ Z_{CI}(s) & Z_{CC}(s) \end{bmatrix} \left\{ \begin{array}{c} < q_I(s) > \\ q_C(s) \end{array} \right\} = \left\{ \begin{array}{c} R_I(s) \\ < 0 > \end{array} \right\} \tag{6.12}$$

where $< >$ denotes a given quantity (the displacement q_I are given and the reaction forces R_I computed). The exact response to an imposed harmonic displacement $q_I(s)$ is given by

$$\{q(s)\} = \begin{bmatrix} I \\ -Z_{CC}^{-1} Z_{CI} \end{bmatrix} \{q_I\} \tag{6.13}$$

The first level of approximation is to use a quasistatic evaluation of this response (evaluate at $s = 0$, that is use $Z(0) = K$). Model reduction on this basis is known as **static or Guyan condensation** [25].

This reduction does not fulfill the requirement of validity over a given frequency range. Craig and Bampton [35] thus complemented the static reduction basis by **fixed interface modes** : normal modes of the structure with the imposed boundary condition $q_I = 0$. These modes correspond to singularities Z_{CC} so their inclusion in the reduction basis allows a direct control of the range over which the reduced model gives a good approximation of the dynamic response.

The Craig-Bampton reduction basis takes the special form

$$\left\{ \begin{array}{c} q_I(s) \\ q_C(s) \end{array} \right\} = \left[\begin{array}{cc} I & 0 \\ -K_{CC}^{-1} K_{CI} & \phi_C \end{array} \right] \{q_R\} \tag{6.14}$$

where the fact that the additional fixed interface modes have zero components on the interface DOFs is very useful to allow direct coupling of various component models. fe_reduc provides a solver that directly computes the Craig-Bampton reduction basis.

A major reason of the popularity of the Craig-Bampton reduction basis is the fact that the interface DOFs q_I appear explicitly in the generalized DOF vector q_R. This is actually a very poor reason that has strangely rarely been challenged. Since the equivalence property tells that the predictions of a reduced model only depend on the projection subspace, it is possible to select the reduction basis and the generalized DOFs independently. The desired generalized DOFs can always be characterized by an observation matrix c_I. As long as $[c_I]\,[T]$ is not rank deficient, it is thus possible to determine a basis $\tilde{T}$ of the subspace spanned by T such that

$$[c_I] \left[\tilde{T}\right] = \left[[I]_{NI\times NI} \quad [0]_{NI\times(NR-NI)} \right] \tag{6.15}$$

The fe_coor function builds such bases, and thus let you use arbitrary reduction bases (loaded interface modes rather than fixed interface modes in particular) while preserving the main interest of the Craig-Bampton reduction basis for coupled system predictions (see example in section 6.2.1).

6.1.6 Substructuring

Substructuring is a process where models are divided into components and component models are reduced before a coupled system prediction is performed. This process is known as **Component Mode Synthesis** in the literature. Ref. [32] details the historical perspective while this section gives the point of view driving the *SDT* architecture (see also [36]).

One starts by considering disjoint components coupled by interface component(s) that are physical parts of the structure and can be modeled by the finite element method. Each component corresponds to a dynamic system characterized by its I/O behavior $H_i(s)$. Inputs and outputs of the component models correspond to interface DOFs.

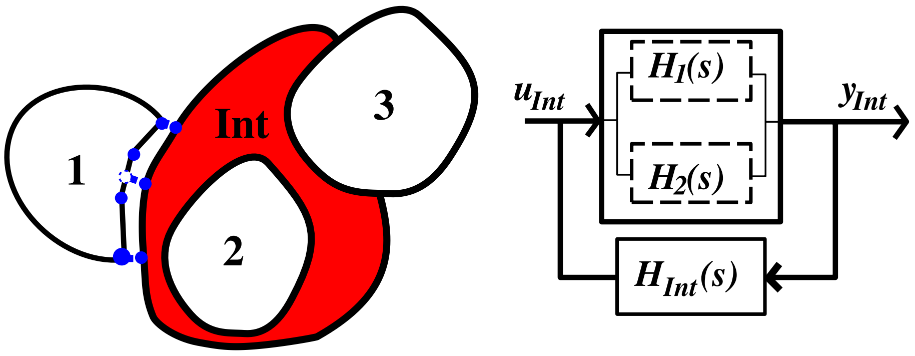


Figure 6.2: CMS procedure.

Traditionally, interface DOFs for the interface model match those of the components (the meshes are compatible). In practice the only requirement for a coupled prediction is that the interface DOFs linked to components be linearly related to the component DOFs $q_{j\mathtt{int}} = [c_j]\,[q_j]$. The assumption that the components are disjoint assures that this is always possible. The observation matrices c_j are Boolean matrices for compatible meshes and involve interpolation otherwise.

Because of the duality between force and displacement (reciprocity assumption), forces applied by the interface(s) on the components are described by an input shape matrix which is the transpose of the output shape matrix describing the motion of interface DOFs linked to components based on component DOFs. Reduced component models must thus be accurate for all those inputs. CMS methods achieve this objective by keeping all the associated constraint or attachment modes.

Considering that the motion of the interface DOFs linked to components is imposed by the components, the coupled system (closed-loop response) is simply obtained adding the dynamic stiffness of the components and interfaces. For a case with two components and an interface with no internal DOFs, this results in a model coupled by the dynamic stiffness of the interface

$$\left(\begin{bmatrix} Z_1 & 0 \\ 0 & Z_2 \end{bmatrix} + \begin{bmatrix} c_1^T & 0 \\ 0 & c_2^T \end{bmatrix} [Z_{\mathtt{int}}] \begin{bmatrix} c_1 & 0 \\ 0 & c_2 \end{bmatrix}\right) \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = [b]\,\{u(s)\} \tag{6.16}$$

The traditional CMS perspective is to have the dimension of the interface(s) go to zero. This can be seen as a special case of coupling with an interface stiffness

$$\left(\begin{bmatrix} Z_1 & 0 \\ 0 & Z_2 \end{bmatrix} + \begin{bmatrix} c_1^T & 0 \\ 0 & c_2^T \end{bmatrix} \frac{\begin{bmatrix} I & -I \\ -I & I \end{bmatrix}}{\epsilon} \begin{bmatrix} c_1 & 0 \\ 0 & c_2 \end{bmatrix}\right) \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = [b]\{u(s)\} \tag{6.17}$$

where ϵ tends to zero. The limiting case could clearly be rewritten as a problem with a displacement constraint (generalized kinematic or Dirichlet boundary condition)

$$\begin{bmatrix} Z_1 & 0 \\ 0 & Z_2 \end{bmatrix} \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = [b]\{u(s)\} \quad \text{with} \quad [c_1 \quad -c_2] \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = 0 \tag{6.18}$$

Most CMS methods state the problem this way and spend a lot of energy finding an explicit method to eliminate the constraint. The *SDT* encourages you to use `fe_coor` which eliminates the constraint numerically and thus leaves much more freedom on how you reduce the component models (see section 6.2.1 and section 6.2.2).

In particular, this allows a reduction of the number of possible interface deformations [36]. But this reduction should be done with caution to prevent locking (excessive stiffening of the interface).

6.1.7 Reduction for parameterized problems

Methods described up to now, have not taken into account the fact that in (6.2) the dynamic stiffness can depend on some variable parameters. To apply model reduction to a variable model, the simplest approach is to retain the low frequency normal modes of the nominal model. This approach is however often very poor even if many modes are retained. Much better results can be obtained by taking some knowledge about the modifications into account [37].

In many cases, modifications affect a few DOFs: $\Delta Z = Z(\alpha) - Z(\alpha_0)$ is a matrix with mostly zeros on the diagonal and/or could be written as an outer product $\Delta Z_{N\times N} = [b_I] \left[\Delta \hat{Z}\right]_{NB\times NB} [b_I]^T$ with NB much smaller than N. An appropriate reduction basis then combines nominal normal modes and static responses to the loads b_I

$$T = \left[\phi_{1...NR} \quad \left[\hat{K}\right]^{-1} [b_I]\right] \tag{6.19}$$

In other cases, you know a typical range of allowed parameter variations. You can combine normal modes are selected representative design points to build a multi-model reduction that is exact at these points

$$T = [\phi_{1...NR}(\alpha_1) \quad \phi_{1...NR}(\alpha_2) \quad ...] \tag{6.20}$$

If you do not know the parameter ranges but have only a few parameters, you should consider a model combining modeshapes and modeshape sensitivities [38] (as shown in the gartup demo)

$$T = \left[\phi_{1...NR}(\alpha_0) \quad \frac{\partial \phi_{1...NR}}{\partial \alpha} \quad ...\right] \tag{6.21}$$

For a better discussion of the theoretical background of fixed basis reduction for variable models see Refs. [37] and [38].

6.2 CMS examples

The *SDT* gives you simple access to all traditional component mode synthesis methods. The following sections, corresponding to the d_cms and d_cms2 demos treat CMS at a low level or using superelements supported by fesuper and fe_super. The later approach is more general in particular because it allows the use of generic and parameterized superelements.

6.2.1 Component mode synthesis

This section complements the d_cms demonstration which discusses classical CMS for the simple example of two stiffened plates shown below. This is meant to get into the details of how to do component mode synthesis by hand. Superelements discussed in section 6.2.2 typically provide cleaner code and easier access to CMS methods.

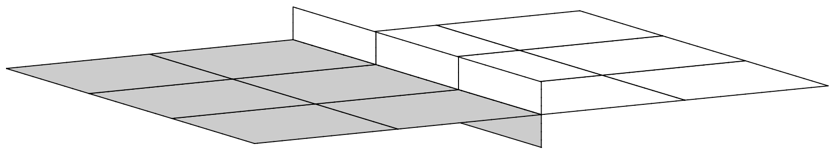


Figure 6.3: CMS example : 2 stiffened plates.

The model is divided in two element groups which have nodes in common

```
IntNode = femesh('find node group1 & group2');
```

For this demo one will use the full set of DOFs

```
mdof = femesh('finddof group1:2');
```

even if each component model has non-zero displacements on its own DOFs only.

```
femesh('selgroup1');
model1=struct('Node',FEnode,'Elt',FEel0,'DOF',mdof,'pl',pl,'il',il);
[m1,k1,mdof]=fe_mk(model1,'options',[0 1]);
tc1=fe_reduc('static',m1,k1,mdof,IntNode);
[md1,f1]=fe_eig(m1,k1,[4 20],mdof,fe_c(mdof,IntNode,'dof',2));
T1 = [tc1 md1(:,find(f1<1.5*fmax))];
```

selects the elements of group 1 (selected elements are stored in the global variable `FEel0`), assembles the component model (the `[0 1]` option of `fe_mk` ensures that DOFs of component 2, that have no stiffness in this case, are retained in the model), computes the static response to imposed unit displacements at interface nodes, computes the normal modes `md1` with interface nodes fixed and keeps those that have a frequency below 1.5 times the maximum frequency of interest (a rule of thumb that usually works well). The basis `T1` which combines **constraint modes** and **fixed interface modes** corresponds to the Craig-Bampton reduction discussed in section 6.1.5.

While using the same reduction for the second component is the traditional solution, the *SDT* allows the use of arbitrary hybrid bases. One can thus consider

```
femesh('selgroup2');
model2=struct('Node',FEnode,'Elt',FEel0,'DOF',mdof,'pl',pl,'il',il);
[m2,k2,mdof]=fe_mk(model2,'options',[0 1]);
ta2=fe_reduc('flex',m2,k2,mdof,IntNode);
[md2,f2]=fe_eig(m2,k2,[4 20 1e3]);
T2 = [ta2 md2(:,find(f2<1.5*fmax))];
```

which combines the flexible response to unit loads applied at the interface (**attachment modes**) with free-interface modes (as used in the MacNeal and Rubin CMS methods).

For a coupled prediction, one just needs to build a basis combining `T1` and `T2` but verifying displacement continuity at the interface nodes. To do so, one builds the observation matrix associated with the interface DOFs

```
cint = fe_c(mdof,IntNode);
```

and finds the a basis of the kernel of `cint*T1-cint*T2` using `fe_coor`

```
T = [T1 (T2-cint'*(cint*T2))]*fe_coor(cint*[T1 -T2]);
```

In the present case, the same interface DOFs are shared by the two components (interface nodes were not duplicated). One must thus eliminate the interface contribution of one of the two components which is done using `(T2-cint'*(cint*T2))`. You could also use `T2b` computed with

```
T2b = T2; T2b(fe_c(mdof,IntNode,'ind'),:)=0;
T = [T1 T2b]*fe_coor(cint*[T1 -T2]);
```

The `d_cms` demo repeats the same prediction with duplicated interface nodes, which is actually cleaner when using the *SDT*. The preferred method remains however the use of superelements as discussed in section 6.2.2.

Given `T` a reduction basis verifying the continuity constraint, the coupled prediction is simply obtained by projection

```
[mdr,fr]=fe_eig(T'*(m1+m2)*T,T'*(k1+k2)*T,[2 20 1e3]);mdr = T*mdr;
```

but you should prefer

```
[mdr,fr]=fe_norm(T,m1+m2,k1+k2);
```

which will eliminate nearly collinear vectors cleanly when `fe_eig` may not. Eliminating collinear vectors may be mandated in applications involving non standard bases (`T=[TA ModeFix]` or [37]) or dealing with problems involving many interface DOFs (plate and solid interfaces).

You can verify with the demo that the coupled prediction is very good up to the sixth flexible mode at 330 rd/s (which is much higher than `fmax=200`).

For component 2, **residual attachment modes** could easily be obtained using

```
md2=md2(:,find(f2<1.5*fmax)); tra2 = ta2 - md2*(md2'*m2*ta2);
```

You can easily verify that they are mass and stiffness orthogonal to the normal modes `md2` (use `[norm(md2'*m2*tra2) norm(md2'*k2*tra2)]`) and that the basis `[tra2 md2(:,find(f2<1.5*fmax))]` gives the same coupled prediction than the one with the attachment modes.

6.2.2 Substructuring using superelements

Superelements can be used to manipulate reduced versions of a full order component model. This application is demonstrated in `d_cms2`. The usual steps for this use of superelements would be to

- assemble the full order model

  ```
  [m,k,mdof]=fe_mk(nodeS,eltS,pl,il);
  ```

- compute a reduction basis. For example the Craig-Bampton basis (constraint and fixed interface modes) associated to the interface DOFs `idof`

  ```
  [T,sdof] = fe_reduc('CraigBampton 20',m,k,mdof,idof);
  ```

- transform to a basis with the first vectors being linked to the interface DOFs and others to superelement DOFs (with the combination constraint mode + fixed interface modes, this is already true) and form the associated superelement DOF vector `sdof`

  ```
  c = fe_c(mdof,idof);
  T = T*fe_coor(c*T,2);
  sdof=[idof;-1-[1:size(T,2)-length(idof)]'/1000];
  ```

- declare the component model as a superelement called `sub1`, define reduced matrices, and save superelement to a file

  ```
  fesuper('set sub1 dof',mdof);
  fesuper('set sub1 k 1 1',k); % matrix 1 is type 1 (stiffness)
  fesuper('set sub1 k 2 2',m); % matrix 2 is type 2 (mass)
  fesuper('set sub1 tr',tr,mdof,sdof)
  fesuper('save FileName sub1');
  ```

  You are now ready to use `sub1` as a standard element. Any model containing an element group called `sub1` will assemble the reduced mass and stiffness models.

- you can associate each of the declared matrices to a non-unit weighting by declaring an element property row `il` with values `il(`*j*`,2:end)` being the weighting coefficients. Thus

  ```
  fesuper('set sub1 ProID 99');
  [mr,kr,sdof]=fe_mk(FEnode,[Inf abs('sub1')],[],[99 1.0 3.0]);
  ```

  sets the `ProID` of the `sub1` superelement to 99, and the `fe_mk` assembly call returns the nominal reduced stiffness and the reduced mass multiplied by 3.

The `fesuper save` and `load` commands let you perform the reduction once for many reuses of the superelement. Note that generic superelements also let you perform a single reduction for a series of components with identical material and geometric properties. Varying material and geometric properties can also be treated (see section 6.3 and Ref. [39]).

6.3 Model parameterization with `upcom`

6.3.1 Theoretical framework

Different major applications use families of structural models. *Update problems*, where a comparison with experimental results is used to update the mass and stiffness parameters of some elements or element groups that were not correctly modeled initially. *Structural design problems*, where component properties or shapes are optimized to achieve better performance. *Non-linear problems* where the properties of elements change as a function of operating conditions and/or frequency (viscoelastic behavior, geometrical non-linearity, etc.).

A *family of models* is defined (see [37] for more details) as a group of models of the general second order form (2.1) where the matrices composing the dynamic stiffness depend on a number of *design parameters* p

$$[Z(p,s)] = \left[M(p)s^2 + C(p)s + K(p)\right] \tag{6.22}$$

Moduli, beam section properties, plate thickness, frequency dependent damping, node locations, or component orientation for articulated systems are typical p parameters. The dependence on p parameters is often very non-linear. It is thus often desirable to use a model description in terms of other parameters α (which depend non-linearly on the p) to describe the evolution from the initial model as a linear combination

$$[Z(p,s)] = \sum_{j=1}^{NB} \alpha_j(p)\,[Z_{j\alpha}(s)] \tag{6.23}$$

with each $[Z_{j\alpha}(s)]$ having constant mass, damping and stiffness properties.

Plates give a good example of p and α parameters. If p represents the plate thickness, one defines three α parameters: t for the membrane properties, t^3 for the bending properties, and t^2 for coupling effects.

p parameters linked to elastic properties (plate thickness, beam section properties, frequency dependent damping parameters, etc.) usually lead to low numbers of α parameters so that the α should be used. In other cases (p parameters representing node positions, configuration dependent properties, etc.) the approach is impractical and p should be used directly.

As for nominal models, parameterized models can be reduced by projection on a constant reduction basis T leading to input/output models of the form

$$\begin{array}{c}\left[T^T Z(p,s) T\right]\{q_R\} = \left[T^T b\right]\{u(s)\} \\ \{y(s)\} = [cT]\{q_R\}\end{array} \quad (6.24)$$

or, using the α parameters,

$$\begin{array}{c}\sum_{j=1}^{NB}\alpha_j(p)\left[T^T \Delta Z_{j\alpha}(s) T\right]\{q_R\} = \left[T^T b\right]\{u(s)\} \\ \{y(s)\} = [cT]\{q_R\}\end{array} \quad (6.25)$$

6.3.2 upcom parameterization for full order models

Although superelements can deal with arbitrary models of the form (6.23), the upcom interface is designed to allow easier parameterization of models. This interface stores a long list of mass M^e and stiffness K^e matrices associated to each element and provides, through the assemble command, a fast algorithm to assemble the full order matrices as weighted sums of the form

$$[M(p)] = \sum_{j=1}^{NE}\alpha_k(p)\left[M_k^e\right] \qquad [K(p)] = \sum_{j=1}^{NE}\beta_k(p)\left[K_k^e\right] \quad (6.26)$$

where the nominal model corresponds to $\alpha_k(p) = \beta_k(p) = 1$.

The basic parameterizations are mass p_i and stiffness p_j coefficients associated to element selections e_i, e_j leading to coefficients

$$\begin{array}{l}\alpha_k, \beta_k = 1 \quad \text{for} \quad k \notin e_i \\ \alpha_k = p_i \quad \text{for} \quad k \in e_i \\ \beta_k = p_j \quad \text{for} \quad k \in e_j\end{array} \quad (6.27)$$

Only one stiffness and one mass parameter can be associated with each element. The element selections e_i and e_j are defined using upcom ParStackAdd commands. In some upcom commands, one can combine changes in multiple parameters by defining a matrix dirp giving the p_i, p_j coefficients in the currently declared list of parameters.

Typically each element is only associated to a single mass and stiffness matrix. In particular problems, where the dependence of the element matrices on the design parameter of interest is non-linear and yet not too complicated, more than one submatrix can be used for each element.

In practice, the only supported application is related to plate/shell thickness. If p represents the plate thickness, one defines three α, β parameters: t for the membrane properties, t^3 for the bending properties, and t^2 for coupling effects. This decomposition into element submatrices is implemented by specific element functions, q4up and q8up, which build element submatrices by calling quad4 and quadb. Triangles are supported through the use of degenerate quad4 elements.

Element matrix computations are performed before variable parameters are declared. In cases where thickness variations are desired, it is thus important to declare which group of plate/shell elements may have a variable thickness so that submatrices will be separated during the call to fe_mk. This is done using a call of the form upcom('set nominal t *GroupID*',FEnode,FEel0,pl,il).

6.3.3 Getting started with upcom

Basic operation of the upcom interface is demonstrated in gartup.

The first step is the selection of a file for the superelement storage using upcom('load FileName'). If the file already exists, existing fields of Up are loaded. Otherwise, the file is created.

If the results are not already saved in the file, one then computes mass and stiffness element matrices (and store them in the file) using

```
upcom('setnominal',FEnode,FEelt,pl,il)
```

which calls fe_mk. You can of course eliminate some DOFs (for fixed boundary conditions) using a call of the form

```
upcom('setnominal',FEnode,FEelt,pl,il,[],adof)
```

At any time, upcom info will printout the current state of the model: dimensions of full/reduced model (or a message if one or the other is not defined)

```
'Up' superelement (stored in '/tmp/tp425896.mat')

Model Up.Elt with 90 element(s) in 2 group(s)
Group 1 :    73 quad4  MatId 1 ProId 3
Group 6 :    17 q4up  MatId 1 ProId 4

Full order (816 DOFs, 90 elts, 124 (sub)-matrices, 144 nodes)
Reduced model undefined
No declared parameters
```

In most practical applications, the coefficients of various elements are not independent. The upcom par commands provide ways to relate element coefficients to a small set of design variables. Once parameters defined, you can easily set parameters with the parcoef command (which computes the coefficient associated to each element (sub-)matrix) and compute the response using the upcom compute commands. For example

```
upcom('load GartUp');
upcom('parstackreset')
upcom('parstackadd k','Tail','group3');
upcom('parstackadd t','Constrained Layer','group6');
upcom('parcoef',[1.2 1.1]);
upcom('info')
cf=upcom('plotelt')
cf.def(1)=upcom('computemode full 6 20 1e3 11')
fecom('scd.3');
```

6.3.4 Reduction for variable models

The upcom interface allows the simultaneous use of a full and a reduced order model. For any model in a considered family, the full and reduced models can give estimates of all the *qualities* (static responses, modal frequencies, modeshapes, or damped system responses). The reduced model estimate is however much less numerically expensive, so that it should be considered in iterative schemes.

The selection of the reduction basis T is essential to the accuracy of a reduced family of models. The simplest approach, where low frequency normal modes of the nominal model are retained, very often gives poor predictions. For other bases see the discussion in section 6.1.7.

A typical application (see the gartup demo), would take a basis combining modes and modeshape sensitivities, orthogonalize it with respect to the nominal mass and stiffness (doing it with fe_norm ensures that all retained vectors are independent), and project the model

```
 upcom('parcoef',[1 1]);
 [fsen,mdsen,mode,freq] = upcom('sens mode full',eye(2),7:20);
 [m,k]=upcom('assemble');T = fe_norm([mdsen mode],m,k);
 upcom('par red',[T])
```

In the gartup demo, the time needed to predict the first 20 modes is divided by 10 for the reduced model. For larger models, the ratio is even greater which really shows how much model reduction can help in reducing computational times.

Note that the projected model corresponds to the currently declared variable parameters (and in general the projection basis is computed based on knowledge of those parameters). If parameters are redefined using `ParStack` commands, you must thus project the model again.

6.3.5 Predictions of the response using `upcom`

The `upcom` interface provides optimized code for the computation, at any design point, of modes (`ComputeMode` command), modeshape sensitivities (`SensMode`), frequency response functions using a modal model (`ComputeModal`) or by directly inverting the dynamic stiffness (`ComputeFRF`). All predictions can be made based on either the full or reduced order model. The default model can be changed using `upcom('OptModel[0,1]')` or by appending `full` or `reduced` to the main command. Thus

```
upcom('ParCoef',[1 1]);
[md1,f1] = upcom('compute mode full 105 20 1e3');
[md2,f2] = upcom('compute mode reduced');
```

would be typical calls for a full (with a specification of the `fe_eig` options in the command rather than using the `Opt` command) and reduced model.

Warning: unlike `fe_eig`, `upcom` typically returns frequencies in Hz (rather than rd/s) as the default unit option is `11` (for rd/s use `upcom('optunit22')`)

Given modes you could compute FRFs using

```
IIxh = nor2xf(freq,0.01,mode'*b,c*mode,IIw*2*pi);
```

but this does not include a static correction for the inputs described by `b`. You should thus compute the FRF using (which returns modes as optional output arguments)

```
[IIxh,mode,freq] = upcom('compute modal full 105 20',b,c,IIw);
```

This approach to compute the FRF is based on modal truncation with static correction (see section 6.1.3). For a few frequency point or for exact full order results, you can also compute the response of the full order model using

```
IIxh = upcom('compute FRF',b,c,IIw);
```

In FE model update applications, you may often want to compute modal frequencies and shape sensitivities to variations of the parameters. Standard sensitivities are returned by the `upcom sens` command (see the *Reference* section for more details).

6.4 Finite element model updating

While the upcom interface now provides a flexible environment that is designed for finite element updating problems, integrated methodologies for model updating are not stabilized. As a result, the *SDT* currently only intends to provide an efficient platform for developing model updating methodologies. This platform has been successfully used, by SDTools and others, for updating industrial models, but the details of parameter selection and optimization strategies are currently only provided through consulting services.

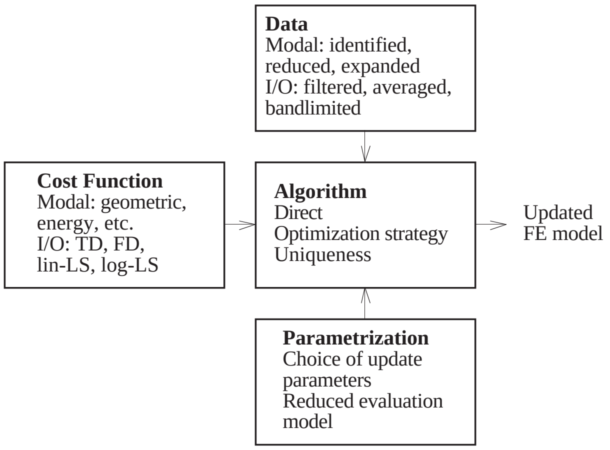

Data
Modal: identified, reduced, expanded
I/O: filtered, averaged, bandlimited
Cost Function
Modal: geometric, energy, etc.
I/O: TD, FD, lin-LS, log-LS
Algorithm
Direct
Optimization strategy
Uniqueness
Updated FE model
Parametrization
Choice of update parameters
Reduced evaluation model

Figure 6.4: FE updating process.

The objective of finite element updating is to estimate certain design parameters (physical properties of the model) based on comparisons of test and analysis results. All the criteria discussed in section 4.2 can be used for updating.

The correlation tools provided by fe_sens and fe_exp are among the best existing on the market and major correlation criteria can easily be implemented. With *SDT* you can thus easily implement most of the existing error localization algorithms. No mechanism is however implemented to automatically translate the results of this localization into a set of parameters to be updated. Furthermore, the updating algorithms provided are very basic.

6.4.1 Error localization/parameter selection

The choice of design parameters to be updated is central to FE update problems. Update parameters should be chosen based on the knowledge that they have not been determined accurately from initial component tests. Whenever possible, the actual values of parameters should be determined using refined measurements of the component properties as the identifiability of the parameters is then clear. If such refined characterizations are not possible, the comparison of measured and predicted responses of the overall system provide a way to assess the probable value of a restricted set of parameters.

Discrepancies are always expected between the model and test results. Parameter updates made based on experimentally measured quantities should thus be limited to parameters that have an impact on the model that is large enough to be clearly distinguished from the expected residual error. Such parameters typically are associated to connections and localized masses.

In practice with industrial models, the FE model is initially divided into zones with one mass/stiffness parameter associated with each zone. The `femesh` `findelt` commands can greatly help zone definition.

Visualizing the strain/kinetic energy distribution of modeshapes is a typical way to analyze zones where modifications will significantly affect the response. The `gartup` demo shows how the strain energy of modeshapes and displacement residuals can be used in different phases of the error localization process.

6.4.2 Update based on frequencies

As illustrated in `demo_fe`, once a set of update parameters chosen, you should verify that the proper range is set (see `min` and `max` values in section 6.3.3), make sure that `Up.copt` options are appropriately set to allow the computation of modes and sensitivities (see `upcom copt` commands), and define a sensor configuration matrix `sens` using `fe_sens`.

With test results typically stored in poles `IIpo` and residues `IIres` (see section 3.3), the update based on frequencies is then simply obtained by a call of the form

```
i2=1:8;  % indices of poles used for the update
[coef,md1,f1] = up_freq('basic',IIpo(i2,:),IIres(i2,:).',sens);
```

The result is obtained by a sensitivity method with automated matching of test and analysis modes using the MAC criterion. A non-linear optimization based solution can be found using up_ifreq but computational costs tend to prevent actual use of this approach. Using reduced order models (see section 6.3.4 and start use with upcom('opt model 1')) can alleviate some of the difficulties but the sensitivity based method (up_freq) is clearly better.

6.4.3 Update based on FRF

An update algorithm based on a non-linear optimization of the Log-Least-Squares cost comparing FRFs is also provided with up_ixf. The call to up_ixf takes the form

```
 coef = up_ixf('basic',b,c,IIw,IIxf,indw)
```

Using up_min for the optimization you will have messages such as

```
Step size: 1.953e-03
     Cost       Parameter jumps ...
  3.9341e-01  -9.83e+00   4.05e+00
```

which indicate reductions in the step size (Up.copt(1,7)) and values of the cost and update parameters at different stages of the optimization. With Up.copt(1,2) set to 11 you can follow the evolution of predictions of the first FRF in the considered set. The final result here is shown in the figure where the improvement linked to the update is clear.

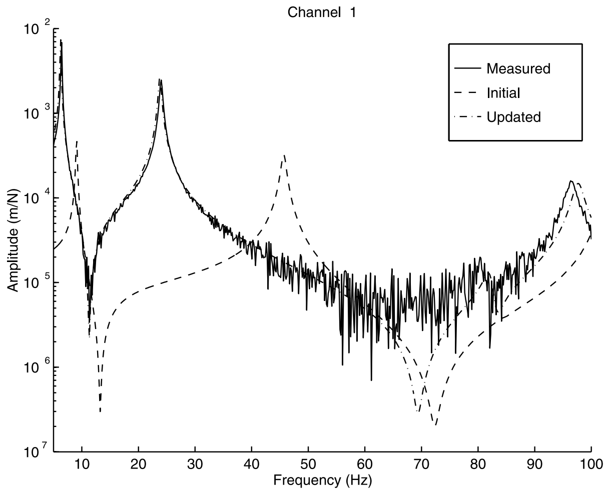


Figure 6.5: Updated FRF.

This algorithm is not very good and you are encouraged to use it as a basis for further study.

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.

7.1 Nodes

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 5.1.5.

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 5.1) and the mesh handling utility `femesh`.

The **only restriction** applied to the `NodeID` is that they should be positive integers, smaller than `round((2^31-1)/100)` $\approx$ `21e6` (this limit is linked to the use of sparse routines for DOF reindexing operations).

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.

7.2 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 5.1).

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 8 (or `doc('eltfun')` from the command line) .

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 7.14.

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).

$\boxed{\text{The general format for } \textbf{element rows} \text{ is}}$

`[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 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`, ...

7.3 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 Graphical edition and standard scripts see section 5.1.4.

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',1)`
`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`).

7.4 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 Graphical edition and standard scripts see section 5.1.4.

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)`
`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`).

7.5 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` or read from FE code results (see `nasread`, `ufread`).
- **input and output shape matrices** which describe how forces are applied and sensors are placed (see `fe_c`, `fe_load`, `bc` page 26).
- **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` , `xfopt`, and `fe_sens`).

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 (see section 6.2.2). 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.

7.6 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 5.2 and detailed in section 7.7.

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. Used by `fe_sens`.
`.copt`	solver options. For use by `upcom`. 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. Used by `upcom`.
`.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,:)`. See details in the `fe_super` reference.
`.mind`	element matrix indices. Used by `upcom`.
`.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. See `fe_super`.
`.Stack`	A cell array containing optional properties further characterizing a finite element model. See `stack_get` for how to handle the stack.
`.Ref`	Generic coordinate transformation specification. See `fe_super`.
`.tdof`	test DOF field. See `fe_sens`.
`.TR`	projection matrix. See `fe_super`.
`.unit`	main model unit system (see `fe_mat convert` for a list of supported unit systems and the associated two letter codes).
`.wd`	working directory

Currently supported entries in the model 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
`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 edge references (structures with fields `.ID` giving the reference number and `.data` with two columns giving `EltId` and edge number) or the similar face references. Sets can be used to define loaded surfaces.
`pro`	defines an element property entry

Currently used `info` entries are

`EigOpt`	gives real eigenvalue solver options (see `fe_eig`).
`OrigNumbering`	original node numering (associated with `feutil` renumber command). This is useful to bypass the limitation on node numbers which must be less than `2^31/100`
`NewNodeFrom`	integer giving the next `NodeId` to be used when adding nodes to the model (used by some commands of `feutil`).
`Freq`	Frequencies given as a structure with field `.X` with frequency values and `.ID` a integer identifier.
`Omega`	rotation vector used for rotating machinery computations (see `fe_cyclic`).

7.7 FEM case data structure

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.
`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.
`Case.b`	left hand side vectors needed to describe load (using DOFs corresponding to the columns of `T`)

The various cases are then stored in the `.Stack` field of the model data structure (this is done by a call to `fe_case`). Each row is a cell array (in Matlab) or a list (in Scilab) giving `{Type,Name,data}`. Supported stack entries for cases are

- `KeepDof`, `FixDof`, `rigid`, `mpc` are used to impose *fixed boundary conditions and constraints*. `SensDof` entries are used to define sensors. These entries are detailed in `fe_case`;
- `DofLoad`, `DOFSet`, `FVol`, `FSurf` are used by `fe_load` to define loads;
- `par` are used by `upcom` to define physical parameters.

7.8 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

7.9 Curves and data sets

A curve is a `data` structure with fields

`.ID`	identification and type of the curve
`.X`	X-axis data
`.Y`	Y-axis data. If a matrix rows correspond to `.X` values and columns are called *channels*
`.Z`	optionnal Z-axis data. Typically one value per channel.
`.data`	a matrix with one row per channel (column of `.Y`). This is used to store DOF information for responses, pole information for modes, ...
`.xunit`	a cell array with three columns giving `label` the meaning of the x axis, `ulabel` the unit label for the x axis, `lftue` the length, force and temperature unit exponents. Typical fields can be generated with `fe_curve('datatype','Time')`
`.yunit`	same as `.xunit` except that there can be as many rows as channels in the `.Y` data.
`.zunit`	same as `.xunit`.
`.name`	name of the curve
`.type`	`'fe_curve'`
`.unit`	unit system of the curve (see `fe_mat convert`)
`.Interp`	optional interpolation method
`.PlotInfo`	type of plotting

To add a curve to the `model.Stack` cell array, the `data` structure must be introduced in a cell array with the form `{'curve', Name, data}`. `Name` is a string identifying the entry.

7.10 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 `femesh 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 7.13. Node selection tools are described in the next section.

7.11 Node selection

`feutil` 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).

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 `==`).
`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 `<=`).
`x>`*a*	selects nodes such that their *x* coordinate is larger than *a*. `x y z` and the logical operators `>`, `<`, `>=`, `<=`, `==` can be used.
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.

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. The basic commands are

- `[NodeID,node]=feutil(['findnode ...'],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.

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.

7.12 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);`
  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`.

`eltind` is the selected elements indices in the element description matrix. `selelt` is 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`.
`EGID` *i*	finds elements with element group identifier *i*. Operators accepted.
`Facing >` *cos x y z*	finds topologically 2-D elements whos 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. `g` retains inter-group edges. Type `m` retains inter-material edges. Type `p` retains inter-property 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).

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])`.

7.13 Constraint and fixed boundary condition handling

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

$$\begin{aligned} \left[Ms^2+Cs+K\right]\{q(s)\} &= [b]\{u(s)\} \\ \{y(s)\} &= [c]\{q(s)\} \\ [c_{int}]\{q(s)\} &= 0 \end{aligned} \tag{7.1}$$

The linear constraints $[c_{int}]\{q(s)\}=0$ can be integrated into the problem using Lagrange multipliers but the preferred approach here is to eliminate these constraints. This is done by building a basis T for the kernel of the constraint equations

$$\text{range}([T]_{N\times(N-NC)}) = \ker([c]_{NS\times N}) \tag{7.2}$$

and solving problem

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

which is strictly equivalent to solving (7.1).

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 tt 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_mk, fe_load, ...

When defining local displacement bases (non zero value of DID in node column 3), master DOFs are defined in the local coordinate system. As a result, M is expected to be define in the global response system while the projected matrix T^TMT is defined in local coordinates. mpc constraints are defined using the local basis.

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

```
model2 = femesh('test 2bay');
model2=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
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_mk(model2,'options',[0 2 2]);

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

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

```
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 this is supported by the `rigid` function. The following illustrates restitution of a constrained solution on all DOFs

```
% Example of a plate with a rigid edge
femesh(';testquad4;divide 10 10;addsel');
% 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;
model.pl=m_elastic('dbval 100 steel');
model.il=p_shell('dbval 110 Mindlin 5e-2');
```

```
% 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')
```

7.14 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.

7.14.1 General information

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 7.14.2 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.

7.14.2 Writing a new element function

The first step to create a new element is to write a new element function.

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

`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)
`estate`	real (`double`) valued element state information. Nominally each column in `estate` corresponds to the internal state of an element. The `estate` for each group is stored in `Case.GroupInfo{jGroup,5};`.
`integ`	`int32` valued constitutive information. The `integ` for each group is stored in `Case.GroupInfo{jGroup,3};`.
`jElt`	number of the current element in `cEGI`
`jGroup`	number of the current element group (order in the element matrix).
`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`).
`pointers`	one column per element in the current group gives : `pointers(1,jElt)` size of desired output or zero. `pointers(5,jElt)` type of desired output. See the `fe_mk MatType` section for a current list. `pointers(6,jElt)` gives the starting index (first element is 0) of integer options for the current element in `integ`. `pointers(7,jElt)` gives the starting index (first element is 0) of real options for the current element in `constit`. The `pointers` for each group is stored in `Case.GroupInfo{jGroup,2};`.

`dof`

Generic DOF definition vector. This vector 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]`.

`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 in section 7.14.3). If some faces have fewer nodes, the last node should be repeated as needed.

`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 :

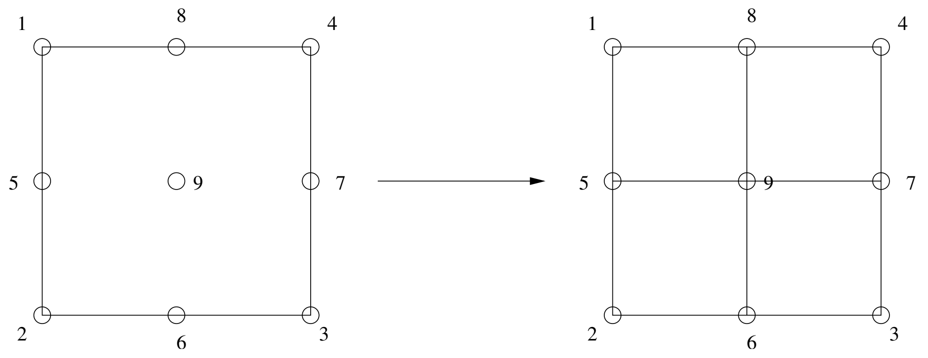


Figure 7.1: Lower order patch representation of a 9 node quadrilateral

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

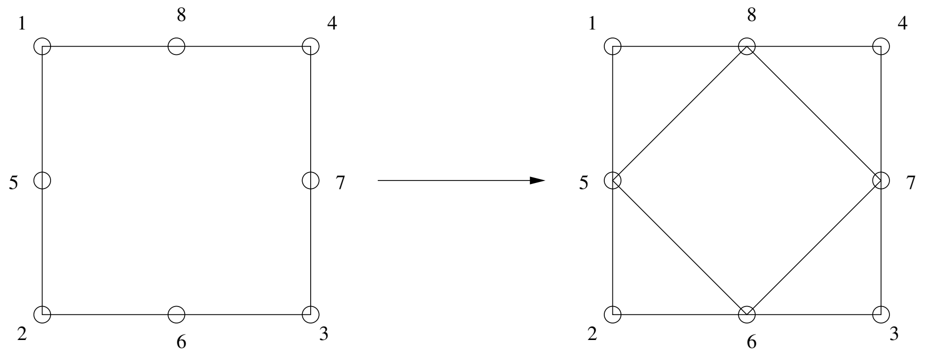


Figure 7.2: Lower order patch representation of a 8 node quadrilateral

integinfo

`[constit,integ,elmap]=elem0('integinfo',pl,il,[MatId ProId])` searches `pl` and `il` for rows corresponding to `MatId` and `ProId` and returns 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.

`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`.

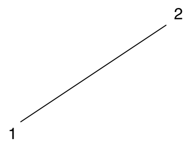
There is currently no automated mechanism to allow users to integrate such computations for their own elements without modifying `fe_load` and `fe_stres`, but this will appear later since it is an obvious maintenance requirement.

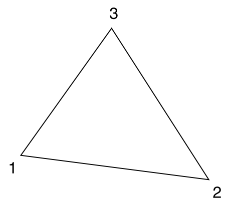
The mechanism that will be used will be similar to that used for matrix assembly. The element function will be required to provide calling formats when called with `elem0('fsurf')` for surface loads, `elem0('fvol')` for volume loads, and `elem0('stress')` for stresses. `fe_load` and `fe_stres` will then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

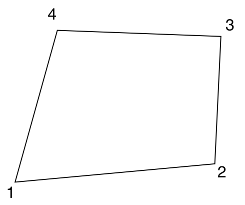
7.14.3 Conventions

Geometric orientation conventions

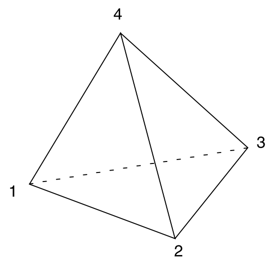
Having chosen the first vertex, each element is defined by:

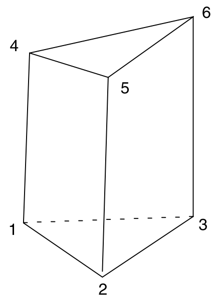


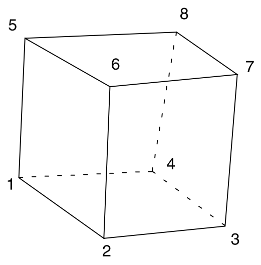




1. The segment:
   • (1) → (2)
2. The triangle: numbering anti-clockwise in the two-dimensional case (in the three-dimensional case, there is no orientation).
   • edge [1]: (1) → (2) (nodes 4, 5, ... if there are supplementary nodes) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (1)
3. The quadrilateral: numbering 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)







4. The tetrahedron: 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)

5. The pentahedron: 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)

6. The hexahedron: 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)

7.15 Generic compiled 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. All the element specific information stored in the `EltConst` data structure.

Second generation volume elements `hexa8b, tetra4b, ...` 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 hexa8b, tetra4b, ...) .
- 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`). For example `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.

Each group of element following this format is characterized by

`integ`	integer constants associated with the group of elements.
`constit`	real valued constants associated with the group. This is where the constitutive law is stored.
`gstate`	group state.
`ElMap`	element matrix map used to distinguish between internal and external element DOF numbering (for example : `hexa8b` 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, ...)
`InfoAtNode`	vector fields at node to be used when assembling the element. No element currently uses this feature.
`EltConst`	element constant information (integration rules, etc.)

7.16 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;`
`j1,j2,j3` ...	loop indices.
`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))`.

Element reference

Element functions supported by *OpenFEM* version 2.0 are listed below. 3-D elements can be degenerated to 2-D by DOF elimination. 2-D elements are assumed in the x-y plane. Plane stress, plane strain or axysimmetry is selected using the element property row in `il`.

Utility elements	
`fe_super`	element function for general superelement support
`integrules`	FEM integration rule support
`fsc`	fluid/structure coupling capabilities

2-D plane stress/strain and axisymmetric elements	
`q4p`	4-node 8-DOF quadrangle
`q5p`	5-node 10-DOF quadrangle
`q8p`	8-node 16-DOF quadrangle
`t3p`	3-node 6-DOF triangle
`t6p`	6-node 12-DOF triangle

2-D isoparametric elements	
`q4pb`	4-node 8-DOF quadrangle
`q5pb`	5-node 10-DOF quadrangle
`q8pb`	8-node 16-DOF quadrangle
`t3pb`	3-node 6-DOF triangle
`t6pb`	6-node 12-DOF triangle

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)

3-D isoparametric solid elements	
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

3-D isoparametric solid elements with non linear gemetric support	
hexa8b	8-node 24-DOF brick
hexa20b	20-node 60-DOF brick
hexa27b	27-node 81-DOF brick
penta6b	6-node 18-DOF pentahedron
penta15b	15-node 45-DOF pentahedron
tetra4b	4-node 12-DOF tetrahedron
tetra10b	10-node 30-DOF tetrahedron

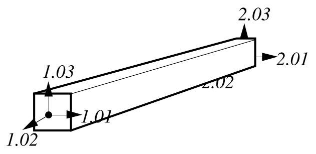
3-D acoustic elements	
flui4	4-node 4-DOF tetrahedron
flui6	6-node 6-DOF pentahedron
flui8	8-node 8-DOF hexahedron

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

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 7.14).

```
[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

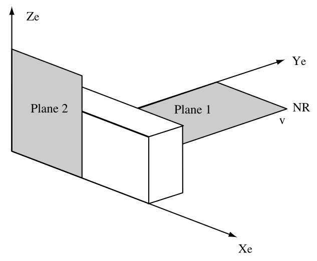
```
[ProID  Type   0  0  0 A]
```

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 (default node is 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**.
`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 :

```
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

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`.

When used, element property rows for `celas` elements take the form (detailed under `p_spring`)

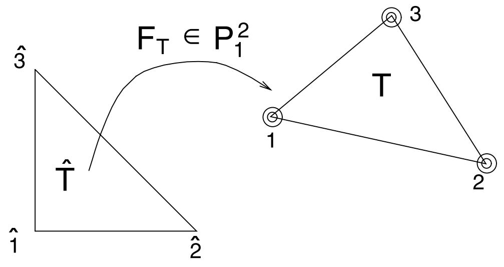
```
[ProID type KvDefault]
```

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

flui4,flui6,flui8

Purpose Isoparametric 4, 6 and 8 node brick fluid elements.

Description The `flui4`, `flui6` and `flui8` elements are isoparametric elements describing linear acoustics. A derivation of these elements can be found in [40].

In a model description matrix, **element property rows** these elements follow the standard format `[n1 ... ni MatID ProId EltId]` (see `elem0`).

The supported material property declaration format is described in `m_elastic` (sub type 2).

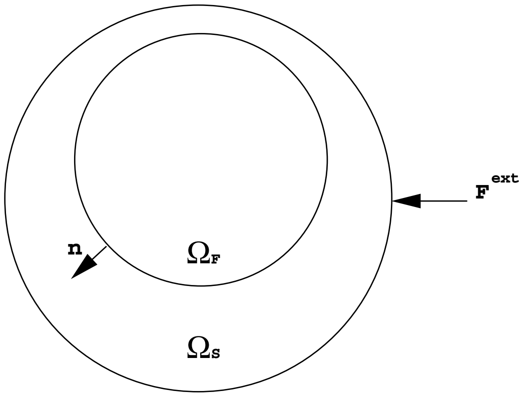
```
[MatId fe_mat('m_elastic',1,2) Rho C eta]
```

See also `m_elastic`, `fe_mat`, `fe_mk`, `feplot` , fsc

fsc

Purpose Non standard element for fluid/structure coupling

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 [41]

$$s^2 \begin{bmatrix} M & 0 \\ C^T & K_p \end{bmatrix} \begin{Bmatrix} q \\ p \end{Bmatrix} + \begin{bmatrix} K(s) & -C \\ 0 & F \end{bmatrix} \begin{Bmatrix} q \\ p \end{Bmatrix} = \begin{Bmatrix} F^{ext} \\ 0 \end{Bmatrix} \tag{8.1}$$

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

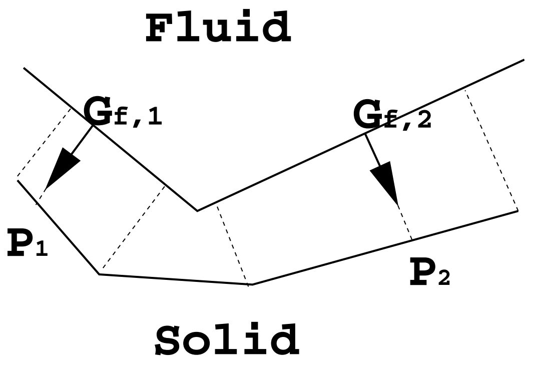
$$\begin{array}{c} \int_{\Omega_S} \sigma_{ij}(u)\epsilon_{ij}(\delta u)dx \Rightarrow \delta q^T K q \\ \int_{\Omega_S} \rho_S u.\delta u dx \Rightarrow \delta q^T M q \\ \frac{1}{\rho_F} \int_{\Omega_F} \nabla p \nabla \delta p dx \Rightarrow \delta p^T F p \\ \frac{1}{\rho_F c^2} \int_{\Omega_F} p \delta p dx \Rightarrow \delta p^T K_p p \\ \int_{\Sigma} p \delta u.n dx \Rightarrow \delta q^T C p \end{array} \tag{8.2}$$

Terms corresponding to the structure are computed using the classical elements of the SDT, and terms corresponding to the fluid are computed using the fluid elements (see `flui4`).

The coupling term C is computed using fluid/structure coupling elements (`fsc` elements).

Only one integration point on each element (the centre of gravity) is used to evaluate C.

When structural and fluid meshes do not match at boundaries, pairing of elements needs to be done. The pairing procedure can be described for each element. For each fluid element F_i, one takes the center of gravity $G_{f,i}$ (see figure), and searches the solid element S_i which is in front of the center of gravity, in the direction of the normal to the fluid element F_i. The projection of $G_{f,i}$ on the solid element, P_i, belongs to S_i, and one computes the reference coordinate r and s of P_i in S_i (if S_i is a quad4, $-1 < r < 1$ and $-1 < s < 1$). Thus one knows the weights that have to be associated to each node of S_i. The coupling term will associate the DOFs of F_i to the DOFs of S_i, with the corresponding weights.



See also `flui4`, `m_elastic`

hexa8, hexa20, penta6, penta15, tetra4, tetra10

Purpose Isoparametric volume elements.

Description The `hexa8` `hexa20` `penta6` `penta15` `tetra4` and `tetra10` elements are the standard isoparametric elements containing DOFs `.01` to `.03` at each node. These elements support 3-D isotropic and orthotropic materials (see `m_elastic`). The newer family of `*b` elements implements the same elements with more options (full anisotropy, geometric non-linearity, integration rules selection, ...).

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]`. For example, the `hexa8` format is `[n1 n2 n3 n4 n5 n6 n7 n8 MatID]`.

Vertex coordinates of the reference element can be found using an `integrules` command containing the name of the element such as `r1=integrules('hexa8');r1.xi`.

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

`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 [40].

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 $k = 1, 7$

See also fe_mat, m_elastic, fe_mk, feplot

hexa8b, hexa20b, hexa27b, penta6b, penta15b, tetra4b, tetra10b

Purpose Isoparametric volume elements with non linear geometric support.

Description This family of elements contains `hexa8b, hexa20b, hexa27b, penta6b, penta15b, tetra4b, tetra10b` which implement standard isoparametric formulations containing DOFs `.01` to `.03` at each node.

In a model description matrix, **element property rows** follow generic format. For an element containing i nodes the format is `[n1 ... ni MatID ProId EltId]`.

Material properties should point the `m_elastic` entries for isotropic, orthotropic or fully anisotropic materials. Hyperelastic material support is not yet properly documented.

Element properties are used to integration rule selection as detailed in `p_solid`.

Vertex coordinates of the reference element can be found using an `integrules` command containing the name of the element such as `r1=integrules('hexa8');r1.xi`.

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

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

$$\int_{\Omega_0} (\rho_0 u'', w) + \int_{\Omega_0} S : \delta e = \int_{\Omega_0} f.dv \quad \forall \quad \delta v \tag{8.3}$$

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

$$F_{i,j} = I + u_{i,j} = \delta_{ij} + \{N_{,j}\}^T \{q_i\} \tag{8.4}$$

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 ($\{e_{11}\ e_{22}\ e_{33}\ e_{23}\ e_{31}\ e_{12}\}$) 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

$$de_{ij} = \left(F^T dF\right)_{Sym} = \left(F_{ki} \left\{N_{,j}\right\}^T \left\{\delta q_k\right\}\right)_{Sym} \tag{8.5}$$

Thus the virtual work of internal loads is given by

$$\int_\Omega S : \delta e = \int_\Omega \left\{\delta q_k\right\}^T \left\{N_{,j}\right\} F_{ki} S_{ij} \tag{8.6}$$

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

$$K_G = \int_\Omega S_{ij} u_{k,i} v_{k,j} + \int_\Omega de : \frac{\partial^2 W}{\partial e^2} : \delta e \tag{8.7}$$

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

$$K_G^e = \int_\Omega \left\{\delta q_m\right\} \left\{N_{,l}\right\} \left(F_{mk} \frac{\partial^2 W}{\partial e^2}_{ijkl} F_{ni} + S_{lj}\right) \left\{N_{,j}\right\} \left\{dq_n\right\} \tag{8.8}$$

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

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

$$S = D : e(u) = \lambda Tr(e) I + 2\mu e \tag{8.9}$$

the building of the constitutive law matrix D is performed in `p_solid BuildConstit` for isotropic, orthotropic and full anisotropic materials.

For a geometric non-linear static computation one will thus solve

$$[K(q^n)] \left\{q^{n+1} - q^n\right\} = \int_\Omega f.dv - \int_{\Omega_0} S : \delta e \tag{8.10}$$

where external forces f are assumed to be non following.

Hyperelasticity For hyperelastic media $S = \partial W / \partial e$ with W the hyperelastic energy. `elem0` currently supports Mooney-Rivlin materials for which the energy takes the form

$$W = C_1(J_1 - 3) + C_2(J_2 - 3) + K(J_3 - 1)^2, \tag{8.11}$$

were (J_1, J_2, J_3) are the reduced invariants of the Green-Cauchy tensor

$$C = I + 2e, \tag{8.12}$$

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

$$J_1 = I_1 I_3^{-\frac{1}{3}}, \quad J_2 = I_2 I_3^{-\frac{2}{3}}, \quad J_3 = I_3^{\frac{1}{2}}, \tag{8.13}$$

where one recals that

$$I_1 = \mathrm{tr}C, \quad I_2 = \frac{1}{2}[(\mathrm{tr}C)^2 - \mathrm{tr}C^2], \quad I_3 = \det C. \tag{8.14}$$

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.

Pour calculer les matrices de rigidit, il suffit donc d'avoir les expressions des drives premires et secondes des invariants par rapport e (ou C), ce qui donne **en repre ON** :

$$\frac{\partial I_1}{\partial C_{ij}} = \delta_{ij}, \quad \frac{\partial I_2}{\partial C_{ij}} = I_1\delta_{ij} - C_{ij}, \quad \frac{\partial I_3}{\partial C_{ij}} = I_3 C_{ij}^{-1}, \tag{8.15}$$

o (C_{ij}^{-1}) dsigne les coefficients de la matrice inverse de (C_{ij}). Pour les drives secondes on a :

$$\frac{\partial^2 I_1}{\partial C_{ij}\partial C_{kl}} = 0, \quad \frac{\partial^2 I_2}{\partial C_{ij}\partial C_{kl}} = -\delta_{ik}\delta_{jl} + \delta_{ij}\delta_{kl}, \quad \frac{\partial^2 I_3}{\partial C_{ij}\partial C_{kl}} = C_{mn}\epsilon_{ikm}\epsilon_{jln}, \tag{8.16}$$

o les ϵ_{ijk} sont dfinis par

$$\begin{cases} \epsilon_{ijk} & = 0 \quad \text{si 2 indices concident} \\ & = 1 \quad \text{si } (i,j,k) \text{ permutation paire de } (1,2,3) \\ & = -1 \quad \text{si } (i,j,k) \text{ permutation impaire de } (1,2,3) \end{cases} \tag{8.17}$$

NB : Si on range les composantes de dformations en colonnes ("engineering strains") sous la forme $(e_{11}, e_{22}, e_{33}, 2e_{12}, 2e_{23}, 2e_{31})'$, les 2 derniers termes de (8.16) correspondent donc respectivement aux 2 matrices

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/2 \end{pmatrix}, \tag{8.18}$$

$$\begin{pmatrix} 0 & C_{33} & C_{22} & 0 & -C_{23} & 0 \\ C_{33} & 0 & C_{11} & 0 & 0 & -C_{13} \\ C_{22} & C_{11} & 0 & -C_{12} & 0 & 0 \\ 0 & 0 & -C_{12} & -C_{33}/2 & C_{13}/2 & C_{23}/2 \\ -C_{23} & 0 & 0 & C_{13}/2 & -C_{11}/2 & C_{12}/2 \\ 0 & -C_{13} & 0 & C_{23}/2 & C_{12}/2 & -C_{22}/2 \end{pmatrix}. \tag{8.19}$$

On utilise finalement les drivations en chane :

$$S = \frac{\partial W}{\partial e} = \sum_k \frac{\partial W}{\partial I_k}\frac{\partial I_k}{\partial e}, \tag{8.20}$$

$$\frac{\partial^2 W}{\partial e^2} = \sum_k \frac{\partial W}{\partial I_k} \frac{\partial^2 I_k}{\partial e^2} + \sum_k \sum_l \frac{\partial^2 W}{\partial I_k \partial I_l} \frac{\partial I_k}{\partial e} \frac{\partial I_l}{\partial e}. \tag{8.21}$$

Attention aux facteurs 2 qui viennent lorsqu'on drive (les invariants) par rapport e au lieu de C.

See section **??**.

integrules

Purpose Command function for FEM integration rule support.

Description This function groups integration rule manipulation utilities used by various elements.

`.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`	$Nnode \times nw(1 + Ndim)$ memory allocation to store the shape functions and their derivatives with respect to physical coordinates $[N\ N,x\ N,y\ N,z]$ (see more details below)
`.jdet`	Nw memory allocation to store the determinant of the jacobian matrix at integration points.
`.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)`)

`hexa8, hexa20, hexa27`

Vertex coordinates of the reference element can be found using the `integrules` command `r1=integrules('hexa8');r1.xi` (or command `'hexa20'`, `'hexa27'`).

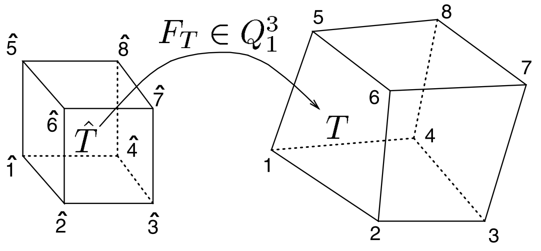


Figure 8.1: `hexa8` reference element.

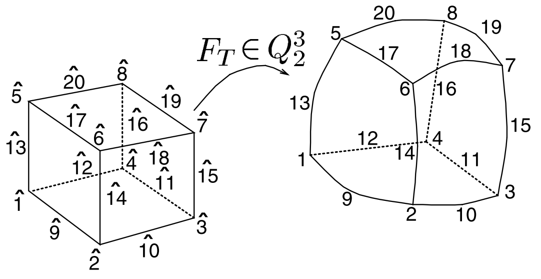


Figure 8.2: `hexa20` reference element.

`penta6, penta15`

Vertex coordinates of the reference element can be found using the `integrules` command `r1=integrules('penta6');r1.xi` (or command `'penta15'`).

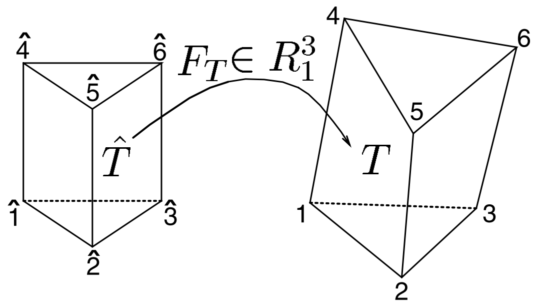
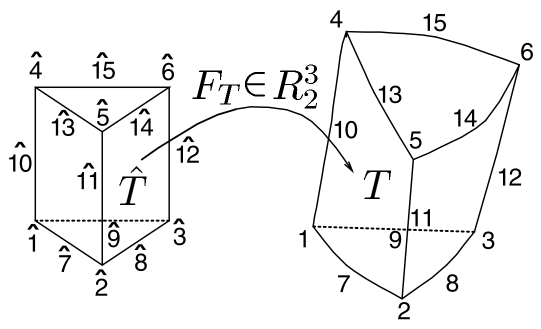


Figure 8.3: `penta6` reference element.



$\hat{4}$ $\hat{15}$ $\hat{6}$ $\hat{13}$ $\hat{5}$ $\hat{14}$ $\hat{10}$ $\hat{11}$ $\hat{T}$ $\hat{12}$ $\hat{1}$ $\hat{9}$ $\hat{3}$ $\hat{7}$ $\hat{8}$ $\hat{2}$
$F_T \in R_2^3$
4 15 6 13 14 10 5 T 12 11 1 9 3 7 8 2

Figure 8.4: `penta15` reference element.

`tetra4,tetra10`

Vertex coordinates of the reference element can be found using the `integrules` command `r1=integrules('tetra4');r1.xi` (command `'tetra10'`).

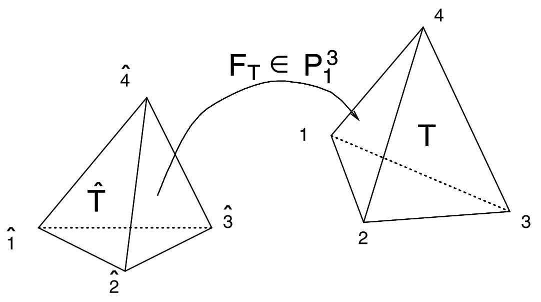


Figure 8.5: `tetra4` reference element.

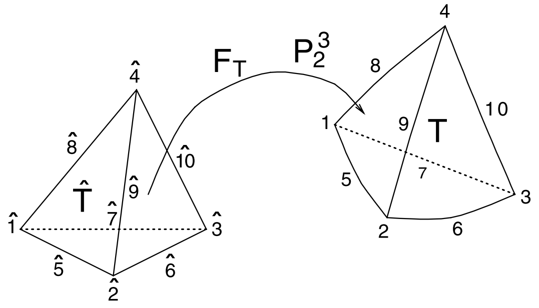


Figure 8.6: `tetra10` reference element.

`q4p, q5p, q8p`

Vertex coordinates of the reference element can be found using the `integrules` command `r1=integrules('quad4');r1.xi`.

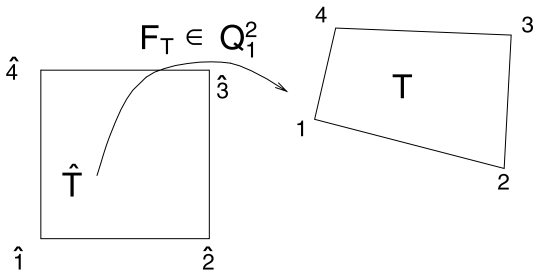


Figure 8.7: `q4p` reference element.

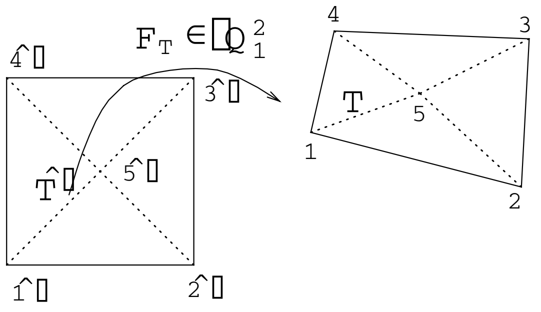


Figure 8.8: q5p reference element.

Vertex coordinates of the reference element can be found using the integrules command `r1=integrules('quadb');r1.xi`.

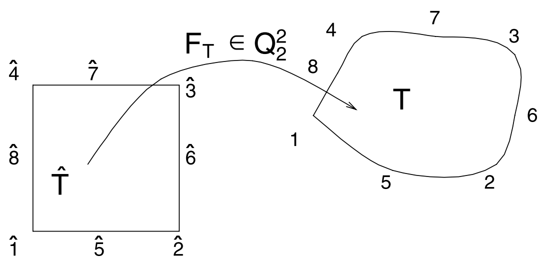


Figure 8.9: q8p reference element.

t3p,t6p

Vertex coordinates of the reference element can be found using the integrules command `r1=integrules('tria3');r1.xi`.

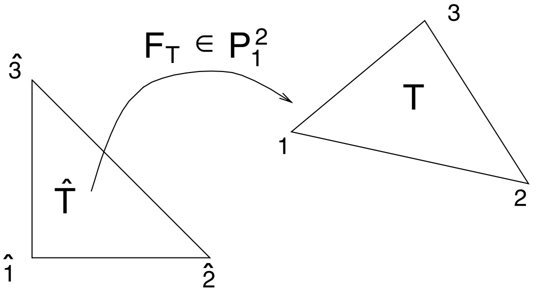


Figure 8.10: `t3p` reference element.

Vertex coordinates of the reference element can be found using the `integrules` command `r1=integrules('tria6');r1.xi`.

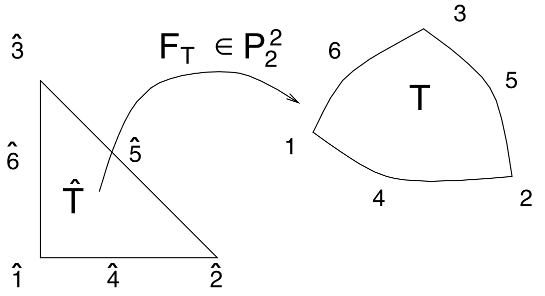


Figure 8.11: `t6p` reference element.

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,node)` where `type` is an `int32` specifies the dimension (2 for 2D, 3 for 3D, 23 for surface in a 3D model), (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.

MatOg elements

The MatOg element family supports a fairly general definition of multi-physic elements whose element integration strategy is fully described by an EltConst data strucure. hexa8b serves as a prototype

```
EltConst=hexa8b('constants',[],[],[]);
integrules('texstrain',EltConst)
EltConst=integrules('stressrule',EltConst);
integrules('texstress',EltConst)
```

Elements of this family are standard element functions (see section 7.14) 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 MatOg family uses an internal DOF sort where each field is given at all nodes sequentially $1x2x...8x1y...8y...$ while the more classical sort by nodes $1x1y...2x...$ is still used for external caccess (and is thus expected to be returned by `DOF = elem('dof')`.

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

$$k^{(e)} = \sum_{ji,jj} \sum_{jw} \left[\{B_{ji}\} \, D_{ji\ jk}(w(jw)) \, \{B_{jj}\}^T \right] J(w(jw))W((jw)) \tag{8.22}$$

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 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 MatOg 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 construction 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

$$[NDN]_{Nnode \times Nw(Ndims+1)} = \left[[N(r,s,t)] \ \left[\frac{\partial N}{\partial x}\right] \ \left[\frac{\partial N}{\partial y}\right] \ \left[\frac{\partial N}{\partial z}\right] \right] \tag{8.23}$$

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

For each strain vector type, one defines an `int32` matrix `EltConst.StrainDefinition{j` 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.

To help you check the validity of a given rule, you should fill the `EltConst.StrainLabels` and `EltConst.DofLabels` fields and use the `integrules('texstrain',EltConst)` command to generate a LATEX printout of the rule you just generated.

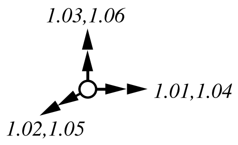
It is assumed that at any integration point, the strain is a function of shape functions and their derivatives.

See also `elem0`0

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])
```

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

$$\begin{bmatrix} M & & & & \text{symmetric} & \\ & M & & & & \\ & & M & & & \\ & & & I_{11} & & \\ & & & -I_{21} & I_{22} & \\ & & & -I_{31} & -I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} \int \rho dV & & & & \text{symmetric} & \\ & M & & & & \\ & & M & & & \\ & & & \int \rho(x^2+y^2)dV & & \\ & & & -I_{21} & I_{22} & \\ & & & -I_{31} & -I_{32} & I_{33} \end{bmatrix}$$

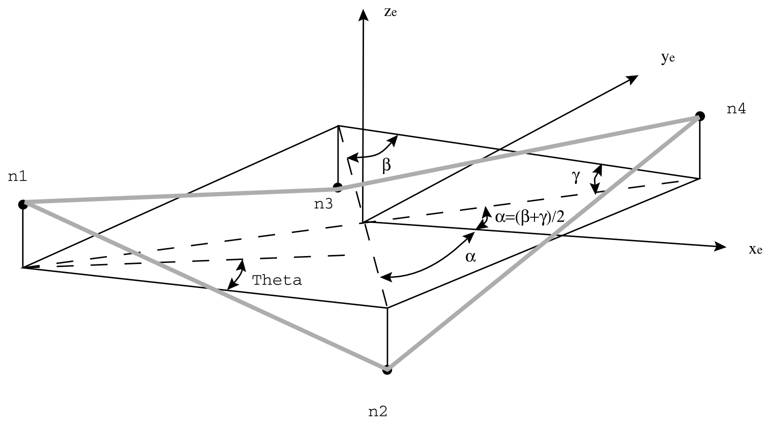
Note that local coordinates `CID` are not currently supported by `mass2` elements.

See also `femesh`, `feplot`

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 (currently unused), 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.

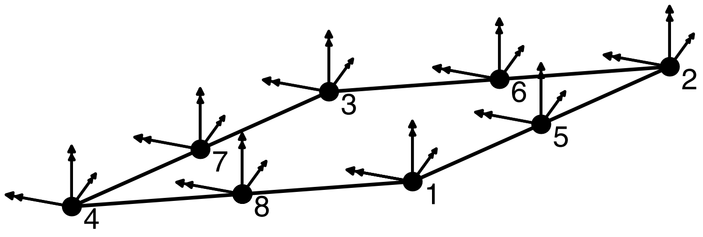
quad4

Supported formulations (il(3) see p_shell) are

quad4, quadb, mitc4

- 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.

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.

mitc4

The MITC4 element is still in a testing phase. It uses 5 DOFs per node with the to 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.

See also m_elastic, p_shell, fe_mk, feplot

q4p, q5p, q8p, t3p, t6p

Purpose 2-D plane stress/strain and axisymmetric elements.

Description 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`.

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 (currently unused)

These elements support isotropic and 2-D anisotropic materials declared with a material entry described in `m_elastic`. Element property declarations follow the format

```
[ProID  Typ  f  N]
```

Where

`Typ`	identifier obtained with `fe_mat('p_shell','SI',1)`
`f`	Formulation : `0` plane stress, `1` plane strain, `2` axisymmetric.
`N`	Fourier coefficient for axisymmetric formulations

The xy plane is used with displacement DOFs `.01` and `.02` given at each node.

The following subsections give more details about the actual formulations of each element. Element matrix calls are handled by the element function itself, while load computations are handled by `fe_load`.

`q4p (plane stress/strain)`

If `n3` and `n4` are equal, the `t3p` element is automatically used in place of the `q4p`.

The `q4p` element uses the `et*2q1d` routines for plane stress and plane 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, q5p, q8p, t3p, t6p

q4p (axisymmetric)

The q4p element uses the et*aq1d routines for axisymmetry.

The radial u_r and axial u_z displacement are bilinear functions over the element.

The coordinates of the reference element vertices are identical to the plane case.

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)

There are five nodes for this incompressible quadrilateral element, four nodes at the vertices and one at the intersection of the two diagonales.

The displacement (u,v) varies linearly within each of the four triangles.

The q4p element uses the et*5noe routines for axisymmetry.

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)

The q8p element uses the et*2q2c routines for plane stress and plane strain and et*aq2c for axisymmetry.

The displacement (u,v) quadratic functions of (x,y) over the element. Strains and stresses are linear functions.

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)

The q8p element uses the et*aq2c routines for axisymmetry.

The radial u_r and axial u_z displacement are quadratic functions.

The coordinates of the reference element vertices are identical to the plane case.

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)

The t3p element uses the et*2p1d routines for plane stress and plane strain.

The displacement (u,v) are assumed to be linear functions of (x,y) (*Linear Triangular Element*), thus the strain are constant (*Constant Strain Triangle*).

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)

The t3p element uses the et*ap1d routines for axisymmetry.

In the triangular cross-section shape of element shown below, a linear polynomial is used to define the radial and axial components of the displacement. Each of the three nodes at the vertices of the triangle has two degrees of freedom (the displacements in the radial and axial directions).

The coordinates of the reference element vertices are identical to the plane case.

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)

The t6p element uses the et*2p2c routines for plane stress and plane strain.

The displacement (u,v) are assumed to be quadratic functions of (x,y) (*Quadratic Triangular Element*), thus the strain are linear (*Linear Strain Triangle*).

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)

The t6p element uses the et*ap2c routines for axisymmetry.

The radial u_r and axial u_z components of the displacements are assumed to be quadratic functions of (u_r, u_z)

The coordinates of the reference element vertices are identical to the plane case.

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 $b_3 = \frac{1+\sqrt{\frac{3}{5}}}{2} = 0.8872985$

See also `fe_mat`, `fe_mk`, `feplot`

q4pb, q8pb, t3pb, t6pb

Purpose 2-D plane stress/strain with integration rule selection.

Description This family of elements implement the same formulations at the non `*b.m` elements with the same names. The interest is mostly linked to the ability to select integration rules with `p_solid` element properties.

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`.

In a model description matrix, **element property rows** for this elements follow the standard format

```
[n1 ... ni MatID ProID EltID]
```

giving the node identification numbers `n1,...ni`, material `MatID`, property `ProID`. Other **optional** information is `EltID` the element identifier.

These elements support isotropic and 2-D anisotropic materials declared with a material entry described in `m_elastic`. Element properties are used to integration rule selection as detailed in `p_solid`.

The xy plane is used with displacement DOFs `.01` and `.02` given at each node.

q9a

Purpose Plane axisymmetric elements with Fourier support.

Description In a model description matrix, **element property rows** for `q9a` elements follow the standard format

```
[n1 ... n9 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)

These elements support isotropic materials declared with a material entry described in `m_elastic`. Element property declarations follow the format

```
[ProID  Typ  f  N]
```

`Typ`	identifier obtained with `fe_mat('p_shell','SI',1)`
`f`	Formulation : `2` axisymmetric.
`N`	Fourier coefficient for axisymmetric formulations

The xy plane is used with displacement DOFs `.01`, `.02` and `.03` given at each node.

The `q9a` element uses the `e*aq2c` to generate matrices.

rigid

Purpose Non-standard element function for the handling of 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

$$\left\{\begin{array}{c} u \\ v \\ w \\ r_x \\ r_y \\ r_z \end{array}\right\}_S = \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & z_{MS} & -y_{MS} \\ 0 & 1 & 0 & -z_{MS} & 0 & x_{MS} \\ 0 & 0 & 1 & y_{MS} & -x_{MS} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \left\{\begin{array}{c} u \\ v \\ w \\ r_x \\ r_y \\ r_z \end{array}\right\}_M$$

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).

This function is then only used for low level access. High level use of rigid links through the use of cases is illustrated in section 7.13 which discusses handling of linear constraints in general.

If coordinate systems are defined in field `Up.bas` (see `basis`), `PID` (position coordinate system) and `DID` (displacement coordinate system) declarations in columns 2 and 3 of `Up.Node` are properly handled.

- `rigid` skips non rigid elements, while `fe_mk` skips rigid elements. A single model description matrix can thus contain definitions for both. `feplot` shows rigid elements as bars.

- 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`.

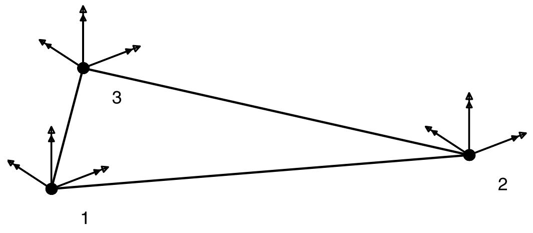
See also Section 7.13, `celas`

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 [42] for example).

tria6 is currently supported for plotting only.

See also

quad4, quadb, fe_mat, p_shell, m_elastic, fe_mk, feplot

9

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 1.3.

Element functions are detailed in chapter s*eltfun.

A list of demonstrations is given in section 1.1.

User Interface (UI) and Graphical User Interface (GUI) Tools	
fecom	UI command function for deformations created with feplot
femesh	UI command function for mesh building and modification
feplot	GUI for 3-D deformation plots
fesuper	UI commands for superelement manipulations
idcom	UI commands for standard identification procedures
idopt	manipulation of identification options
iicom	UI commands for measurement data visualization
ii_mac	GUI for MAC and other vector correlation criteria
iiplot	GUI for the visualization of frequency response data
xfopt	UI to manipulate database wrappers

Experimental Model Identification	
idcom	UI commands linked to identification
idopt	manipulation of options for identification related functions
id_rc	broadband pole/residue model identification
id_rcopt	alternate optimization algorithm for id_rc
id_rm	minimal and reciprocal MIMO model creation
id_nor	optimal normal mode model identification
id_poly	weighted least square orthogonal polynomial identification
id_dspi	direct system parameter identification algorithm
ii_poest	narrow-band single pole model identification
ii_pof	transformations between pole representation formats
psi2nor	optimal complex/normal mode model transformation
res2nor	simplified complex to normal mode residue transformation

UI and GUI Utilities	
comgui	general purpose functions for the graphical user interfaces
commode	general purpose parser for UI command functions
comstr	general purpose string handling routine
iimouse	mouse related callbacks (zooming, info, ...)
feutil	mesh handling utilities
ii_plp	overplot vertical lines to indicate pole frequencies
setlines	line style and color sequencing utility

Frequency Response Analysis Tools	
db	amplitude in dB (decibels)
ii_cost	FRF comparison with quadratic and logLS cost
ii_mmif	Multivariate Mode Indicator Function
phaseb	phase (in degrees) with an effort to unwrap along columns
rms	Root Mean Square response

Test/analysis correlation tools	
fe_exp	experimental shape expansion
fe_sens	sensor configuration declaration and sensor placement tools
ii_comac	obsolete (supported by ii_mac)
ii_mac	GUI for MAC and other vector correlation criteria

Finite Element Analysis Tools	
fe2ss	methods to build ss models from full order FEM
fe_c	DOF selection and I/O matrix creation
fe_case	Cases (loads, boundary conditions, etc.) handling
fe_ceig	computation and normalization of complex modes
fe_coor	transformation matrices for Component Mode Synthesis
fe_eig	partial and full eigenvalue computations
fe_load	assembly of distributed load vectors
fe_mat	material property handling utilities
fe_mk	assembly of full and reduced FE models
fe_norm	orthonormalization and collinearity check
fe_reduc	utilities for finite element model reduction
fe_stres	element energies and stress computations
fe_super	generic element function for superelement support
rigid	projection matrix for linearized rigid body constraints

Model Format Conversion	
nor2res	normal mode model to complex mode residue model
nor2ss	assemble state-space model linked to normal mode model
nor2xf	compute FRF associated to a normal mode model
qbode	fast computation of FRF of a state-space model
res2ss	pole/residue to state space model
res2tf	pole/residue to/from polynomial model
res2xf	compute FRF associated to pole/residue model
ss2res	state-space to pole/residue model

Finite Element Update Tools	
upcom	user interface for finite element update problems
up_freq	semi-direct update by comparison modal frequencies
up_ifreq	iterative update by comparison of modal frequencies
up_ixf	iterative update based on FRF comparison
up_min	minimization algorithm for FE update algorithms

Interfaces with Other Software	
ans2sdt	reading of ANSYS binary files (FEMLink)
nasread	read from MSC/NASTRAN .dat, .f06, .o2, .o4 files (some with FEMLink)
naswrite	write data to MSC/NASTRAN bulk data deck (some with FEMLink)
nas2up	extended reading of NASTRAN files
ufread	read Universal File Format (some with FEMLink)
ufwrite	write Universal File Format (some with FEMLink)

Other Utilities	
basis	coordinate transformation utilities
ffindstr	find string in a file
order	sorts eigenvalues and eigenvectors accordingly
remi	integer rem function (remi(6,6)=6 and not 0)
setlines	line type and color sequencing
sdth	*SDT* handle objects
ofact	creation and operators on ofact matrix objects

ans2sdt

Purpose Interface between ANSYS and SDT (part of FEMLink)

Syntax

```
ans2sdt('BuildUp FileName')
def=ans2sdt('def FileName.rst')
```

Description

`BuildUp`

`ans2sdt('BuildUp FileName')` reads the binary files `FileName.rst` for model defition and `FileName.emat` for element matrices. The result is stored in the global variable `Up` (a type 3 superelement handled by `upcom`). `FileName.mat` is used to store the superelement.

For recent versions of ANSYS, you may have to manually add the `ematwrite,yes` command to the input file to make sure that all element matrices are written. This command is not accessible from the ANSYS menu.

`Def`

`def=ans2sdt('def FileName.rst')` or `def=ans2sdt('def FileName.mode')` reads deformations in .rst or .mode files

A basic application is the display of an ANSYS solution with

```
model=ans2sdt('buildup test');         % read model
def=ans2sdt('def test.rst');           % read deformations
cf=feplot; cf.model=model; cf.def=def; % display
```

See also `nasread`, `ufread`, section 5.5.1

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
```

`[nodeGlobal,bas]= basis(nodeLocal,bas)`

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`.

basis

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.

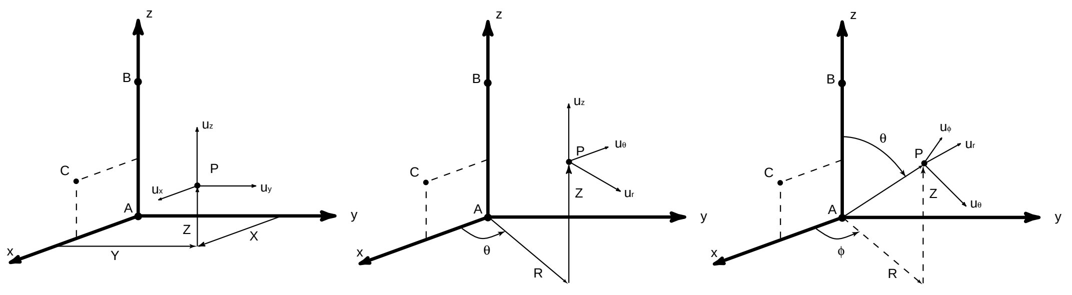


Figure 9.1: Coordinates convention.

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]',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`.

`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)`.

`[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 7.1,section 7.2
Note : the name of this function is in conflict with `basis` of the *Financial Toolbox*.

commode

Purpose General purpose command parser for user interface command functions.

Syntax

```
commmode('CommandFcn','ChainOfCommands')
```

Description Most user interfaces operations in the *Structural Dynamics Toolbox* are grouped in UI command functions (`iicom`, `idcom`, `fecom`, `femesh`, etc.). The desired commands and possible options are passed to the command functions as text commands.

Conventions used in the helps to specify string commands used by user interface functions are

italic	standard names for numerical or string values
`()`	same as italic but used for the on-line rather than HTML help
`[c1,c2]`	alternatives for a command (separated by commas)

Thus `ch[,c] [`*i*`,+,-,+`*i*`,-`*i*`]` means that `ch 14`, `chc 12:14`, `chc+`, `ch-2` are all valid commands. Commands are text strings so that you can use `fecom ch[1 4]` or `fecom('ch 1 4')` but not `fecom ch 1 4` where `ch`, `1` and `4` are interpreted by Matlab as 3 separate strings.

When building complex commands you may need to compute the value used for variable. Some commands actually let you specify an additional numeric argument (`feplot('textnode',[1 2 3])` and `feplot('textnode 1 2 3')` are the same) but in other cases you will have to build the string yourself using calls of the form `feplot(['textnode' sprintf(' %i',[1 2 3])])`

The UI command functions only accept one command at a time, so that `commode` was introduced to allow

- *command mode*: replace the Matlab prompt `>>` by a `CommandFcn>` which directly sends commands to the command function(s).
- *command chaining*: several commands separated by semi-columns `;`. The parsing is then done by `commode`.
- *scripting*: execute all commands in a file.

Most command functions send a command starting by a `';'` to `commode` for parsing. Thus `commode ('iicom','cax1;abs')` is the same as `iicom (';cax1;abs')`

The following commands are directly interpreted by `commode` (and not sent to the command functions)

`q,quit`	exits the command mode provided by `commode` but not MATLAB
`script FName`	reads the file `FName` line by line and executes the lines as command strings

The following syntax rules are common to `commode` and MATLAB

`%comment`	all characters after a `%` and before the next line are ignored
`[]`	brackets can be used to build matrices
`;`	separate commands (unless within brackets to build a matrix)

See also `comstr`, `iicom`, `fecom`, `femesh`

comstr

Purpose String handling functions for the *Structural Dynamics Toolbox*.

Syntax See details below

Description The user interfaces of the *Structural Dynamics Toolbox* have a number of string handling needs which have been grouped in the `comstr` function. The appropriate formats and usual place of use are indicated below.

`istrue=comstr(Cam,'string')`

String comparison. `1` is returned if the first characters of `Cam` contain the complete `'string'`. `0` is returned otherwise. This call is used extensively for command parsing. See also `strncmp`.

`[opt,CAM,Cam]=comstr(CAM,'string','format')`

Next string match and parameter extraction. `comstr` finds the first character where `lower(CAM)` differs from `string`. Reads the remaining string using the `sscanf` specified `format`. Returns `opt` the result of `sscanf` and `CAM` the remaining characters that could not be read with the given format.

`[opt,CAM,Cam]=comstr(CAM,'string','%c')` is used to eliminate the matching part of `string`.

`[CAM,Cam] = comstr(CAM,ind)`

Command segmentation with removal of front and tail blanks. The first `ind` characters of the string command in capitals `CAM` are eliminated. The front and tail blanks are eliminated. `Cam` is a lowercase version of `CAM`. This call to `comstr` is used in all UI command functions for command segmentation.

`opt = comstr(CAM,[-1 default])`

Option parameter evaluation. The string `CAM` is evaluated for numerical values which are output in the row vector `opt`. If a set of default values `default` is given any unspecified value in `opt` will be set to the default.

`date = comstr(CAM,[-3])`

Return the standard date string. Used by `ufwrite`, `naswrite`, etc. See also `date`, `datenum`.

`CAM = comstr(CAM,[-4 nc ])`

Fills the string `CAM` with blanks up to `nc` characters.

`comstr(Matrix,[-5 fid],'format')`

Formatted output of `Matrix`, the `format` is repeated as many times as `Matrix` has columns and a formatted output to `fid` (default is `1` standard output). For example you might use `comstr(ii_mac(md1,md2)*100,[-5 1],'%6.0f')`.

`st1=comstr(st1,-7,'string')`

Used for dynamic messaging on the command line. On UNIX platforms (the backspace does not work properly on Windows), the string `st1` is erased before `'string'` is displayed.

`comstr(tt,-17,'type')`

This is used to generate tabular output of the cell array `tt` to various supported types : `excel` (Microsoft Excel only available on windows), `html`, `csv` (comma separated values, readable by excel), `tex`.

See also `commode`

db, phaseb

Purpose

Compute the decibel magnitude.
Compute the unwrapped phase in degrees.

Syntax

```
m = db(xf)
p = phaseb(xf)
```

Description

`db` computes the decibel magnitude of each element of the matrix `xf`. An equivalent would be

```
 m = 20*log10(abs(xf))
```

`phaseb` is an extension to the case of multiple FRF stacked as columns of a matrix `xf` of the `phase` routine available in the *System Identification Toolbox*. It computes the phase in **degrees** with an effort to keep the phase continuous for each column.

Example

Here is an example that generates the two FRF of a SIMO system and plots their magnitude and phase.

```
 a=[0 1;-1 -.01];b=[0;1];c=[1 0;0 1];d=[0;0];
 w=linspace(0,2,100)'; xf=qbode(a,b,c,d,w);
 clf;
 subplot(211);plot(w,dbsdt(xf));    title('dB magnitude')
 subplot(212);plot(w,phaseb(xf));title('Unwrapped phase in degrees')
```

See also

The `xf` format, `iiplot`

fe2ss

Purpose Build state-space or normal mode form from FE model.

Syntax

```
[sys,basis] = fe2ss('command',MODEL,CASE,C)
[sys,basis] = fe2ss('command',m,k,mdof,b,rdof,C,c)
[nor,basis] = fe2ss( ... ,'nor')
```

Description `fe2ss` is meant to allow users to build state-space (see section 2.4) and normal mode models (see section 2.2) from full order model matrices. The procedure is always decomposed in the following steps

- call `fe_reduc` build a reduction basis `T` (see section 6.1). This usually includes a call to `fe_eig` with options ***`EigOpt`*** provided in the `fe2ss` command
- call `fe_norm` to orthonormalize `T` with respect to mass and stiffness (obtain a model in the normal mode form (2.4), see section 2.2) and eliminate collinear vectors if any
- call `nor2ss` or project model matrices depending on the number of outputs

The `basis` output argument is given so that the user can call `nor2ss` repeatedly without computing the basis again. This is in particular useful for changes in the sensor configuration which have no effect on the retained basis.

High level input arguments are a `MODEL` (see section 5.1) and possibly a `CASE` if not defined in the model (see section 5.2). Note that the `CASE` **must** contain load and sensor entries (see `fe_case`). `C` the damping model can be a system damping matrix, a scalar uniform damping ratio or a vector of damping ratios.

Low level input arguments are those of `fe_reduc` with the additional damping and output shape matrix information.

`m`, `k`	symmetric real mass and stiffness matrix
`mdof`	associated DOF definition vector describing DOFs in `m` and `k`
`b`	input shape matrix describing unit loads of interest. Must be coherent with `mdof`.
`bdof`	alternate load description by a set of DOFs (`bdof` and `mdof` must have different length)
`rdof`	contains definitions for a set of DOFs forming an isostatic constraint (see details below). When `rdof` is not given, it is determined through an LU decomposition done before the usual factorization of the stiffness. This operation takes time but may be useful with certain elements for which geometric and numeric rigid body modes don't coincide.
`C`	damping model. Can specify a full order damping matrix using the same DOFs as the system mass `M` and stiffness `K` or a scalar damping ratio to be used in a proportional damping model.
`c`	output shape matrix describing unit outputs of interest (see section 2.1). Must be coherent with `mdof`.

Standard bases used for this purpose are available through the following commands.

`Free [ , Float]` *`EigOpt`*

The standard basis for modal truncation with static correction discussed in section 6.1.3. *`EigOpt`* are `fe_eig` options used to compute the modeshapes (typically `6` *`nm Shift`* for Lanczos with no reordering, *`nm`* number of desired modes, *`Shift`* mass shift for structures with rigid body modes).

Computation of the static correction for structures with rigid body modes is a standard problem discussed in section 6.1.4. `fe2ss` uses the mass-shift value in *`EigOpt`* to select the method. If the shift is zero, it is assumed that the structure has no rigid body modes. If the shift is non-zero, the shifted attachment modes (6.11) are used as a default. You can obtain the standard attachment modes (6.10) by adding the `Float` modifier to the command.

`CraigBampton` *`nm`*

It is really a companion function to `fe_reduc CraigBampton` command. The retained basis combines fixed interface attachment modes and constraint modes associated to DOFs in `bdof`.

This basis is less accurate than the standard modal truncation for simple predictions of response to loads, but is often preferred for coupled (closed loop) predictions.

Example

```
mdl=demosdt('demo ubeam mix');

mdl=fe_case(mdl,'addtocase','SensDof','Out',[343.01 343.02 347.03]')
[sys,T] = fe2ss('free 6 10',mdl,[.01;.02;.03]);
w=linspace(10,1e3,2500);
nor2xf(T,[.01;.02;.03],mdl,w,'hz plot');
```

See also demo_fe, fe_reduc, fe_mk, nor2ss, nor2xf

fecom

Purpose UI command function for the visualization of 3-D deformation plots

Syntax

```
fecom
fecom CommandString
fecom('CommandString',AdditionalArgument)
```

Description `fecom` provides a number of commands that can be used to manipulate 3-D deformation plots are handled by the `feplot`/`fecom` interface. A **tutorial** is given section 5.4. Other examples can be found in `gartfe`, `gartte` and other demos. Details on the interface architecture are given under `feplot`.

This help lists all commands supported by the interface (calling `fecom` or `feplot` is insensitive to the user).

- `cf=feplot` returns a SDT handle to the current `feplot` figure (se details in the `feplot` help). The handle is used to provide simplified calling formats for data initialization and text information on the current configuration. You can create more than one `feplot` figure with `cf=feplot(`*`FigHandle`*`)`.
- without input arguments, `fecom` calls `commode` which provides a command mode for entering different possibly chained `fecom` commands.
- the first input argument should be a string containing a single `fecom` command, or a chain of semi-column separated commands starting with a semi-column (`fecom(';com1;com2')`). Such commands are parsed by `commode`.
- some commands, such as `TextNode`, allow the use of additional arguments

`Anim[,One][,Time,Freq][,col][nCycle` *`i`*`, Start` *`i`*`, Step]`

Deformed structure animation. The animation is not movie based so that you can actively rotate, change mode, ... without delay. The `AnimStep` command is only used when you really want to create movies.

The animation is started/interrupted using the animation button which calls the `anim start` command. You can set animation properties in the `General` tab of the `feplot properties` figure.

To control animation speed and replay you can use `fecom('AnimTime` *`nStep tStep tStart`*`)` which specifies the number of times that you want the animation to run (0 to run continuously), the minimum time spent at each time step (default zero), and the wait time between successive runs of the same animation (default 0, only works with time mode animation). You may need to fix the color limits manually using `cf.ua.clim=[0 1e3]`.

The default animation (use of `AnimFreq` to return to the default) adds a certain phase shift (`2*pi/nCycle`) to the amplification factor of the deformations currently displayed and updates the plot. The default `nCycle` value is obtained using `feplot AnimnCycle25`.

The command `AnimTime` starts the animation in a mode that increments deformations while preserving the amplification. This is appropriate for animation of time responses.

By default `Anim` animates all the axes in the current figure. You can animate, the current axis only by adding `One` anywhere in the `anim` command.

By default `Anim` does not animate colors. `fecom('AnimCol')` sets color animation to dual sided (alternates between a max value and its opposite). You can animate colors without deformations if you define colors for the current selection without defining a deformation.

Animation speed is very dependent on the figure renderer. See the `fecom Renderer` command.

`Anim[Movie` *i*`,Avi]`

Creating a movie. `M=feplot('anim movie 50')` returns a Matlab movie with 50 animation steps.

Starting with Matlab 6.0 you can use `fecom('animavi')` to create a movie. The movie is created using `avifile` commands with defaults in set to `sdtdef('avi')`. You may change the defaults (see `avifile`) using for example

`sdtdef('avi',{'quality',100,'fps',1,'compression','Cinepak'})`

Compressions of `None` gives higest quality. `Indeo3` and `Indeo5` give poor results with colored patches.

`cax`*i*`, ca+`

Change current `axes`. `cax` *i* makes the axis *i* (an integer number) current. `ca+` makes the next axis current.

For example, `fecom(';sub2 1;cax1;show line;ca+;show sensor')` displays a line plot in the first axis and a sensor plot in the second.

See also the `Axes` tab in the `feplot properties` figure and the `iicom sub` command. In particular `SubStep` is used to increment the deformation numbers in each subplot.

`ch[,c] [`*i*`,+,-,+`*i*`,-`*i*`],`

Displayed deformation control. `feplot` is generally used to initialize a number of deformations (as many as columns in `mode`). `ch` *i* selects the deformation(s) *i* to be displayed (for example `ch 1 2` overlays deformations 1 and 2). By default the first deformation is displayed (for line and sensor plots with less than 5 deformations, all deformations are overlaid). You can also increment/decrement using the `ch+` and `ch-` commands or the `+` and `-` keys when the current axis is a plot axis. `ch+`*i* increments by *i* from the current deformation.

You can also select deformations shown in the `Deformations` tab in the `feplot properties` figure.

When using more than one axis (different views or deformations), the `ch` commands are applied to all `feplot` axes while the `chc` commands only apply to the current axis.

The `SubStep` command is useful to obtain different deformations in a series of axes. Thus to display the first 4 modes of a structure you can use : `fecom(';sub 1 1;ch1;sub 2 2 step')` where the `sub 1 1` is used to make sure that everything is reinitialized. You can then see the next four using `fecom('ch+4')`.

For line and sensor plots and multiple channels, each deformation corresponds to an object and is given a color following the `ColorOrder` of the current axis is used. `feplot` line and sensor plots compatible with the use of `setlines` for line type sequences.

`ColorData [,sel`*i*`] [`*Type*`]`

Color definitions Color information is defined for element selections (see the `fecom sel` commands) and should be defined with the selection using a call of the form, `cf.sel(i)={`*`'SelectionString'`*`,'ColorData', ...}`. `fecom('colordata sel`*i*` ...',...)` is the corresponding low level call. Accepted `ColorData` commands are listed below

`Elt`	The low level call `fecom('ColorDataElt',Ener,IndInElt)` specifies element colors in `Ener` and with `IndInElt` row positions in the global element description matrix of the given colors. A typical application is the case of energies with `fe_stres` (see the `d_ubeam` demo). If `Ener` gives the color for all elements, you can omit `IndInElt`. You can also provide `Ener` as a `struct` with fields `.data .IndInElt`, or `.data .EltId`.
`Ener`	`fecom('ColorData EnerK')` calls `fe_stres` to compute the strain energy for the elements in the current selection and deformation set and displays the result as element colors. `EnerM` computes the kinetic energy. The color animation mode is set to `ScaleColorOne`. This requires material and element properties to be defined in the `InitModel` command.
`gmp`	`fecom('ColorDataG')` defines a color for each element group, `m` for each `MatID`, and `p` for each `ProID`. The color animation mode is set to `ScaleColorOne`.
`Node`	low level call to set a color defined at nodes `fecom('ColorData',cmode)` where `cmode` is a `size(node,1)` by `size(mode,2)` matrix defining *nodal colors*. `fecom('ColorDataNode',mode,mdof)` defines nodal colors that are proportional to the norm of the nodal displacement. You can obtain nodal colors linked to the displacement in a particular direction using `i1=fe_c(mdof,.03,'ind');fecom('ColorDataNode', md0(i1,:), mdof(i1))` even though for displacements in the `xyz` directions `fecom('ColorDataZ')` is shorter.
`Stress`	the `ColordataStress`*i* command defines the selection color by calling `fe_stres` with command `Stress`*i*. The color animation mode is set to `ScaleColorOne`. This requires material and element properties to be defined in the `InitModel` command.
`Uniform`	in this mode the deformation/object index is used to define a uniform color following the axis `ColorOrder`.
`xyz,all ...`	`fecom('ColorDataZ')` defines a color that is proportional to motion in the z direction, ... `ColorData19` will select DOF 19 (pressure). The color animation mode is set to `ScaleColorDef`. `fecom('ColorDataALL')` defines a color that is proportional to motion norm.

Note: When displaying results colors are sometimes scaled using the amplification factor used for deformations. Thus, to obtain color values that match your input exactly, you must use the `fecom ScaleColorOne` mode. In some animations you may need to fix the color limits manually using `cf.ua.clim=[0 1e3]`.

`Color [,sel`*i*`] [Edge ..., Face ..., Bar, Legend]`

Default `EdgeColor` and `FaceColor` properties of the different patches can be set to `none`, `interp`, `flat`, `white`, ... using `fecom('ColorEdgeNone')`, ...

`fecom('ColorEdge',ColorSpec)` where `ColorSpec` is any valid MATLAB color specification, is also acceptable.

`EdgeColor` and `FaceColor` apply to the current selection. The optional `Sel`*i* argument can be used to change the current selection before applying the command.

You can also modify the properties of a particular object using calls of the form `set(cf.o(`*i*`),'edgecolor',`*ColorSpec*`)` (see `fecom go` commands and illustrations in `gartte`).

`fecom('colorbar')` calls the MATLAB `colorbar` to display a color scale to the left of the figure. `feplot` updates this scale when you change the deformation shown. You can use `colorbar` commands to modify its position, and `fecom ColorBarOff` commands to reinitialize a subplot without a color bar.

`fecom('ColorLegend')` uses the MATLAB legend command to create a legend for group, material or property colors. Of course, the associated selection must have such colors defined with a `Colordata[M,P,G]` command.

`ga` *i*

`fecom('ga` *i*`')` or `cf.ga(`*i*`)` gets pointers to the associated axes. See details under the same `iicom` command. A typical application would be to set multiple axes to the same view using `iimouse('view3',cf.ga(:))`.

`go` *i*

Get handles to `fecom` objects. This provides and easy mechanism to modify MATLAB properties of selected objects in the plot (see also the `set` command).

For example, `set(fecom('go2'),'linewidth',2)` will use thick lines for `feplot` object 2 (in the current `feplot` axis).

You will probably find easier to use calls of the form `cf=feplot` (to get a handle to the current `feplot` figure) followed by `set(cf.o(2),'linewidth',2)`. If the `feplot` object is associated to more than one MATLAB object (as for text, mixed plate/beam, ...) you can access separate pointers using `cf.o(2,1)`. The `gartte` demo gives examples of how to use these commands.

`Head [,freq, po, fs]`

Set header strings for automated title generation. `fecom ('head',`*`Labels`*`)`, where *`Labels`* should be a string matrix or a cell array of string, can be used to associate a title string to each deformation of the current deformation set.

`fecom('head')` defines default strings. `fecom('headfreq',`*`freq`*`)` creates strings of the form `Mode 1 at 100Hz`. `fecom('headfs',`*`freq`*`)` gives strings of the form `1 @ 100`. `fecom('headpo',`*`po`*`)` gives for poles strings of the form `6.5 Hz 1.00 %`.

`fecom('titopt111')` turns automatic titles on (see `iicom`). `fecom('titopt0')` turns them off.

Note that the `iicom head` commands can be used to place additional titles in the figure. `cf.head` returns a pointer to the header axis. Mode titles are actually placed in the header axis in order to bypass inappropriate placement by MATLAB when you rotate/animate deformations.

`Info`

Displays information about the declared structure and the objects of the current plot in the command window. This info is also returned when displaying the *SDT* `handle` pointing to the `feplot` figure. Thus `cf=feplot` returns

```
cf =
  FEPLOT in figure 2
    Selections: cf.sel(1)='groupall';
                cf.sel(2)='WithNode {x>.5}';
  Deformations: [ {816x20} ]
  Sensor Sets: [ 0 (current 1)]
  Axis 3 objects:
   cf.o(1)='sel 2 def 1 ch 9 ty1'; % mesh
   cf.o(2) % title
```

which tells what data arrays are currently defined and lists `feplot` objects in the current axis. `fecom('pro')` opens the `feplot properties` figure which provides an interactive GUI for `feplot` manipulations.

`InitDef[ , Back]`

Initialization of deformations. You can (re)declare deformations at any point using `cf.def(`*`i`*`)={mode,mdof}` where `cf` a *SDT* handle to the figure of interest and *i* the deformation set you which to modify. With database wrappers, `cf.def(`*`i`*`)=XF(5)` is also acceptable. `cf.def(i)=def` where `def` is a structure with fields `.def`, `.DOF`, `.data` are also accepted calls.

For animation of test results, `mdof` can be given using the 5 column format used to define arbitrary sensor directions in `fe_sens`. Automatic expansion is also supported using `cf.def={def,exp}` as illustrated in the `fe_sens WireExp` command.

`feplot('InitDef',mode,mdof)` is an alternate calling format that defines the current deformation.

`cf.def(i) = {mode,mdof,freq}` or `feplot('InitDef',mode, mdof, freq)` will also display the mode frequency (it calls `fecom head` automatically). In the first calling format, the current deformation is first set to *i*.

`InitDef` updates all axes. `InitDefBack` returns without updating plots.

InitModel

Initialization of structure characteristics. The preferred calling format is `cf.model=model` where the fields of `model` are described in section 7.6. This makes sure that all model information is stored in the `feplot` figure. `cf.mdl` then provides a handle that lets you modify model properties in scripts without calling `InitModel` again.

You can then edit the model graphically with the `Model properties` figure: define and set materials, element properties, and cases; start solutions with its `Simulate` tab, ... In the `feplot properties` figure you can visualize energies or stresses by defining selections with this color information.

Lower level formats are `cf.model={node,elt,bas}`
(or `feplot('InitModel' ,node,elt,bas)` (see `basis` for `bas` format information). `InitModelBack` does not update the plot (you may want to use this when changing model before redefining new deformations).

The command is also called when using `femesh plotelt`, or `upcom plotelt` (which is equivalent to `cf.model=Up`), `fe_sens('plotlinks',sens)`.

InitSens

Initialization of sensors. You can declare sensors independently of the degrees of freedom used to define deformations (this is in particular useful to show measurement sensors while using modeshape expansion for deformations). Sensor and arrow object show the sensor sets declared using `initsens`.

Translation sensors in global coordinates can be declared using a DOF definition vector `cf.sens(i)={mdof}` or `feplot('initsens',mdof)`. In the first calling format, the current sensor set is first set to *i*.

Sensors in other directions are declared by replacing `mdof` by a 5 column matrix following the format

```
SensorID  NodeID  nx ny nz
```

with `SensorID` an arbitrary identifier (often 101.99 for sensor of unknown type at node 101), `NodeID` the node number of the sensor position, `[nx ny nz]` a unit vector giving the sensor direction in global coordinates (see section 4.1).

`fe_sens` provides additional tools to manipulate sensors in arbitrary directions. Examples are given in the `gartte` demo.

`Plot`

`feplot('plot')`, the same as `feplot` without argument, refreshes axes of the current figure. If refreshing the current axis results in an error (which may occasionally happen if you modify the plot externally), use `clf;iicom('sub')` which will check the consistency of objects declared in each axis. Note that this will delete `Text` objects as well as objects created using the `setobject` command.

`Prop`

`feplot('pro')` initializes or refreshes the `feplot` property GUI. You can also use the `Edit:Feplot Properties ...` menu. A description of this GUI is made in section 5.4.

`Renderer[Opengl,zBuffer,Painters][,default]`

This command can be used to swith the renderer used by `feplot`. Animation speed is very dependent on the figure renderer. When creating the figure `fecom` tries to guess the proper renderer to use (`painters`, `zbuffer`, `opengl`), but you may want to change it (using the `Feplot:Render` menu or `set(gcf,'renderer', 'painters')`, ...). `painters` is still good for wire frame views, `zbuffer` has very few bugs but is very slow on some platforms, `opengl` is generally fastest but still has some significant rendering bugs on UNIX platforms.

To avoid crashes when opening `feplot` in OpenGL mode use `cingui('Renderer zbuffer default')` in your MATLAB startup file.

`Scale [ ,Defs, Dofi, equal, match, max, one]`

Automatic deformation scaling. Scaling of deformations is the use of an amplification factor very often needed to actually see anything. A deformation scaling coefficient is associated with each deformed object object. The `Scale` commands let you modify all objects of the current axis as a group.

You can specify either a length associated with the maximum amplitude or the scaling coefficient.

The base coefficient `scc` for this amplification is set using `fecom('ScaleCoef scc')`, while `fecom('ScaleDef scd')` sets the target length. `fecom('scd 0.01')` is an accepted shortcut. If *scd* is zero an automatic amplitude is used. You can also modify the scaling deformation using the `l` or `L` keys (see `iimouse`).

`fecom` supports various scaling modes summarized in the table below. You can set this modes with `fecom('scalemax')` ... commands.

Scaling mode	Scaling of 1st deformation	Scaling of other deformations
`max`	Amplitude of Max DOF set to `scd`.	Amplitude of Max DOF set to `scd`.
`equal`	Amplitude of Max DOF set to `scd`.	Amplitude of other deformations equal to the first one
`match`	Amplitude of Max DOF set to `scd`.	Amplitude of other deformations set to optimize superposition. When using two deformation sets, rather than two modes in the same set, their DOFs must be compatible.
`coef`	Deformation amplitude multiplied by `scd`.	Same as first deformation.
`one`	Sets `scd` to 1 and uses `coef` mode (so further changes to `scd` lead to amplification that is not equal to 1).	Same as first deformation.

Warning : using `ScaleMax` or `AnimFreq` can lead to negative or complex amplification factors which only makes sense for frequency domain shapes.
`fecom('scalecoef')` will come back to positive amplification of each object in the current `feplot` axis.

`ScaleDofi` is used to force the scaling DOF to be *i*. As usual, accepted values for *i* are of the form `NodeID.DofID` (`1.03` for example). If *i* is zero or not a valid DOF number an automatic selection is performed. `ScaleDof` can only be used with a single deformation set.

You can change the `scale` mode using the `FEplot:Scale` menu or in the `Axes` tab of the `feplot properties` figure.

`ScaleColor`

Color animation (see `fecom ColorData` for how these are defined) supports two modes. `ScaleColorOne` does not scale color deformations. `ScaleColorDef` uses the amplification coefficient set for the associated deformation. Once a color is selected, the `axes clim` property can be used to adjust the range.

`Sel [`*`ElementSelectors`*`, GroupAll, Reset]`

Selection of displayed elements. What elements are to be displayed in a given object is based on the definition of a selection (see section 7.12).

The default command is `'GroupAll'` which selects all elements of all element groups (see section 7.2 for details on model description matrices). `cf.sel(1)='Group1 3:5'` will select groups 1, 3, 4 and 5. `cf.sel(1)='Group1 & ProId 2 & WithNode {x>0}'` would be a more complex selection example.

To define other properties associated with the selection (`ColorData`, ...), use a call of the form `cf.sel(i)={`*`'SelectionString','OtherProp',OtherPropData`*`}`.

To return to the default selection use `fecom('SelReset')`.

`SetObject` *`i`* `[,ty` *`j`*`] [,def` *`k`*`] [,ch` *`k`*`] [,sel` *`s`*`, sen` *`s`*`]`

Set properties of object i. Plots generated by `feplot` are composed of a number of objects. 1 (surface view), 2 (wire frame view), 3 (stick view of sensors), 4 (undeformed structure), 5 (node text labels), 6 (DOF text labels), 7 (arrow view of sensors). A printout of objects existing in the current plot and their properties is given when displaying the *SDT* `handle` associated to the `feplot` figure (using `cf=feplot;disp(cf)`).

The `SetObject` command lets you modify these properties and add/remove new objects. Objects are added if the object number does not exist and removed if the declared object type is zero (*j*`=0`).

The deformation number *k* is an index in the deformation(s) currently selected for the plot using the `ch` command.

The elements actually displayed can be specified by giving an existing selection number *s*. For sensor objects, you can specify the sensor set with `sen s` if you don't want to use the current one.

The current axis summary obtained with `cf=feplot` gives the preferred calling format `cf.o(i)='SetObjArgs'` where you give a string with the options associated to the `SetObject` command. The use of these commands is illustrated in the `gartte` demo. Note that you can use the call, to set other MATLAB properties of the created objects. `cf.o(1)={'ty2 def1 ch1 sel1','linewidth',2,'color','r'}` will, for example, create a wire-frame object using thick red lines.

`Show [patch,line,sensor,arrow, ...]`

Basic plots are easily created using the `show` commands which are available in the `FEplot:Show ...` menu).

`patch`	surface view with hidden face removal and possible color coding (initialized by `fecom('ShowPatch')`). `cf.o(1)` object type is `1`. For color coding, see `colordata` commands.
`line`	wire frame plot of the deformed structure (initialized by `fecom('ShowLine')`). `cf.o(2)` object type is `2`.
`sens`	*Sensor plots* with sticks at sensor locations in the direction and with the amplitude of the response (initialized by `fecom('ShowSen')`). `cf.o(2)` object type is `3`.
`arrow`	*Sensor plots* with arrows at sensor locations in the direction and with the amplitude of the response (initialized by `fecom('ShowArrow')`). `cf.o(2)` object type is `7`.
`DefArrow`	Deformation plots with lines connecting the deformed and undeformed node positions. (initialized by `fecom('ShowDef')`). `cf.o(2)` object type is `8`.
`FEM`	only shows FEM element groups for models mixing test and FEM information
`test`	only shows test element groups for models mixing test and FEM information
`links`	shows a standard plots with the test and FEM meshes as well as links used for topological correlation (see `fe_sens`).
`map`	`fecom('ShowMap',MAP)` displays the vector map specified in MAP (see `feutil GetNormalMap`).
`2def`	is used for cases where you want to compare two deformations sets. The first two objects only differ but the deformation set they point to (1 and 2 respectively). A typical call would be `cf.def(1)={md1,mdof,f1}; cf.def(2)={md2,mdof,f2}; fecom('show2def')`.

Once the basic plot created you can add other objects or modify the current list using the `DofText`, `TextNode`, and `SetObject` commands.

`Sub [`*i j*` ], SubIso, SubStep`

Drawing figure subdivision (see `iicom` for more details). This lets you draw more than one view of the same structure in different axes. In particular the `SubIso` command gives you four different views of the same structure/deformation.

`SubStep` or `Sub` *i j* `Step` increments the deformation shown in each subplot. This command is useful to show various modeshapes in the same figure. Depending on the initial state of the figure, you may have to first set all axes to the same channel. Use `fecom('ch1;sub 2 2 step')` for example.

`Text [off, Node [,Select], Dof d]`

Node/DOF text display. `TextOff` removes all text objects from the current `feplot` axis. `TextNode` displays the numbers of the nodes in `FEnode`. You can display only certain node numbers by a node selection command *Select*. Or giving node numbers in `fecom('textnode',i)`.

`TextDOF` displays the sensor node and direction. If DOF definitions *i* are given in the command string or using `fecom('textdof', i)`, only those DOFs are displayed. `TextDOF` only displays the text linked to currently declared sensors so you may want to change those using the `feplot InitSens` command.

`TitOpt [ ,c] i`

Automated title/label generation options. `Titopt i` sets title options for all axes to the value *i*. *i* is a three digit number with units corresponding to `title`, decades to `xlabel` and hundreds to `ylabel`. By adding a `c` after the command (`titoptc 111` for example), the choice is only applied to the current axis.

The actual meaning of options depends on the plot function (see `iiplot`). For `feplot`, titles are shown for a non zero title option and not shown otherwise. Title strings for `feplot` axes are defined using the `fecom head` command.

`Triax [ , On, Off]`

Orientation triax. Orientation of the plotting axis is shown using a small triax. `Triax` initializes the triax axis or updates its orientation. `TriaxOff` deletes the triax axis (in some plots you do not want it to show). Each triax is associated to a given axis and follows its orientation. The triax is initially positioned at the lower left corner of the axis but you drag it with your mouse.

Finally can use `fecom('triaxc')` to generate a triax in a single active subplot.

`Undef [ , Dot, Line]`

Undeformed structure appearance. The undeformed structure is shown as a line which is made visible/invisible using `UnDef`. When visible, the line can show the node locations (use `UnDefDot`) or link nodes with dotted lines (use `UnDefLine`).

`View [...]`

Orientation control. See `iimouse view`.

See also `feplot`, `fe_exp`, `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
FEelt	main finite element model description matrix
FEel0	selected finite element model description matrix
FEel1	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 `FEelt`.

This is an example of the creation of `FEelt` using 2 selections (`FEel0` and `FEel1`)

```
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]`

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`.

`AddTest [,`*`NodeShift`*`,Merge]`

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`.

`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`.

The new elements are given the `EGID -1` so that they will be ignored in model assembly operations with `fe_mk`.

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.

`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`.

`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`. 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 in 4 triangles using the `div2` command

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(';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);
fecom textnode

% Example 3 : you may use uneven division
femesh('testquad4'); % one quad4 created
femesh('divide',[0 .1 .2 1],[0 .3 1]);
femesh plotel0
```

`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 7.12).

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.

`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('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(st);  femesh plotel0

% Example 3 : you may uneven extrusions in z direction
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
```

`FindDof` *`ElementSelectors`*, `GetDof`

Find DOFs used by the elements selected with *`ElementSelectors`*. These are the DOFs that would be used in an assembly of the model using `fe_mk`. For example

`mdof = femesh('find Dof Group 1:3')`

returns the DOF definition vector for all elements in groups 1, 2 and 3. If a model matrix is given as second argument, it is used instead of `FEelt` (additional numeric arguments for element selection can still be provided).

The equivalent `feutil` command is `GetDof`. `mdof=feutil('get DOF',elt,adof)` returns DOFs used in the assembly of the model `elt`. If a selection of active DOFs `adof` is given, `fe_c` is used to determine the model DOFs present in `adof`.

Note that node numbers set to zero are ignored by `feutil` to allow elements with variable number of nodes.

Finally, you can obtain the list in a part of `elt` by specifying the associated element selection in the command string.

Find [Elt,El0] *ElementSelectors*

Find elements based on a number of selectors described in section 7.12. 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.

When operators are accepted, equality and inequality operators can be used. Thus `group~=[3 7]` or `pro < 5` are acceptable commands. This command can be accessed directly with `feutil`. The example above is equivalent to

```
[ind,elt]=feutil('findelt eltid 1:10 ',model)
```

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 7.11.

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'))
```

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

```
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.

GetElemF

Header row parsing. In an element description matrix, element groups are separated by header rows (see section 7.2) 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 FindNode command. This command is equivalent to the feutil call

node=feutil(['findnode ...'],FEnode, FEelt,FEel0).

GetNormal[Elt,Node][,Map]

`[normal,cg]=feutil('getNormal[elt,node]',node,elt)` returns normals to elements/nodes in model `node`, `elt`. `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.
`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`

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('test2bay;plotelt');
femesh('infoelt');   % 2 groups at this step
femesh joingroup1:2  % 1 group now
```

or using elements type

```
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 7.6) 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`.

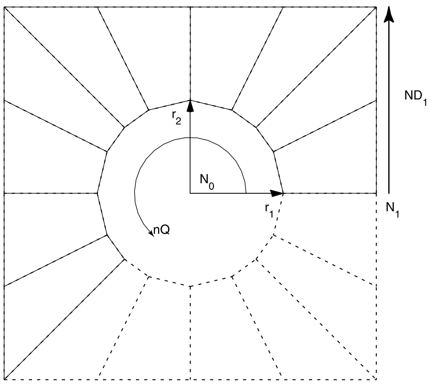
An alternate call is `femesh('ObjectBeamLine',ind)` where `ind` is a vector containing the node numbers. You can also specify a element name other than `beam1` and properties to be placed in columns 3 and more using `femesh('ObjectBeamLine -EltName',ind,prop)`.

`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:
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 1 1',nodes,4,2)` returns model with 4 and 2 divisions in each direction.

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=3, 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[Circle,Cylinder]`

These object constructors follow the format

```
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('object circle 1 1 1 2 0 0 1 30');
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);
feplot(model)
```

`Optim [Model, NodeNum]`

`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`.

`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 section 7.14.3).

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.

`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(';testquad4;divide 2 3;')
FEelt=FEel0;femesh('dividegroup1 withnode1');
model=femesh;
% Orient elements in group 2
model.Elt=feutil('orient 2 n 0 0 -1',model);
```

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`.

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`. `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');
```

`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(';testquad4;divide 2 3');
FEel0 = 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(';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(';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 7.11.

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 FEel0 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 FEel0 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(';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 `FEelt`, 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('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 5.1, demos `gartfe`, `d_ubeam`, `beambar` ...

feplot

Purpose Gateway function for 3-D visualization of structures. See also the companion function `fecom`.

Description `feplot` refreshes all `feplot` axes of the current figure.

`cf=feplot` returns a *SDT* `handle` to the current `feplot` figure. You can create more than one `feplot` figure with `cf=feplot(`*`FigHandle`*`)`.

`cf.model={node,elt}` initializes the FE model displayed in the current figure (see `fecom InitModel`).

`cf.def(`*`i`*`)={def,dof}` initializes a deformation set. `cf.def(`*`i`*`)={def,dof,freq}` where `freq` is a list of frequencies of poles automatically generates title labels for each deformation (see `fecom InitDef`).

`cf.sens(`*`i`*`)={sdof}` initializes a sensor set (see `fecom InitSens`).

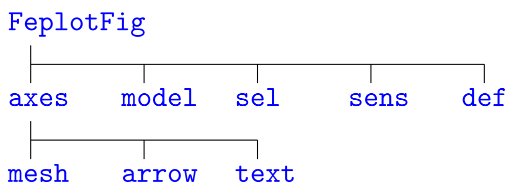
`cf.sel(`*`i`*`)= `*`'ElementSel'`* initializes a selection to use element selected by *`ElementSel`*. Note that you may want to declare color data simultaneously using `cf.sel(`*`i`*`)= {`*`'ElementSel'`*`,'Colordata `*`Command`*`',`*`Args`*`}`.

`cf.o(`*`i`*`)= {`*`'ObjectSpec'`*`,'PatchProperty',PatchValue}` modifies the properties of object *i* in the current `feplot` axis.

A complete list of commands is given under the companion function `fecom` while the rest of this section gives more details on the `feplot` architecture. For a tutorial see section 5.4.

The old format `feplot(node,elt,mode,mdof,2)` is still supported but you are encouraged to switch to the new and more general procedure outlined above.

Views of deformed structures are obtained by combining information from various data arrays that can be initialized/modified at any time. The object hierarchy is outlined below with the first row being data arrays that store information and the second row objects that are really displayed in MATLAB `axes`.



`axes`	describe axes to be displayed within the `feplot` figure. Division of the figure into subplots (MATLAB axes) is obtained using the `iiplot sub` commands. Within each plot, basic displays (wire mesh, surface, sensor, arrow corresponding to `mesh`, `arrow`, or `text` objects) can be obtained using the `fecom show` commands while more elaborate plots are obtained using `fecom setobject` commands. Other axes properties (rotations, animation, deformation selection, scaling, title generation, etc.) can then be modified using `fecom` commands.
`model`	Model data structure (see section 7.6) Initialized using the `InitModel` command (see `fecom`). `cf.mdl` is a handle to the model contained in the `feplot` figure. The model must be defined before any plot is possible.
`sel`	*element selections* describe which elements are displayed. The standard selection displays all elements of all groups. `fecom sel` commands or `cf.sel(i)` let you define selections that only display some elements. See also the `fecom SetObject` commands. Color information is defined for each selection (see `fecom color` commands).
`sens`	*sensor selections* describe sets of sensors. Sensor selections are used to display the response at measurement locations through stick or arrows. Initialized using the `InitSens` command or `cf.sens(i)` calls (see `fecom`).
`def`	*deformation sets* describe deformations at a number of DOFs. Initialized using the `InitDef` command or `cf.def(i)` calls (see `fecom`).

Objects

`mesh`

`mesh` objects represent a deformed or undeformed finite element mesh. They are used both for wire-frame and surface representations. `mesh` objects are characterized by indices giving the element selection, deformation set, channel (deformation number), and color type. They can be modified using calls or the form

```
cf = feplot; % get sdth object handle
cf.o(2) = 'sel 1 def 1 ch 3'
```

or equivalently with `fecom setobject` commands. `fecom show` commands resets the object list of the current axis.

Each `mesh` object is associated to up to three MATLAB `patch` objects associated respectively with real surfaces, segments and isolated nodes. You can access individual pointers to the `patch` objects using `cf.o(i,j)` (see `fecom go` commands).

`arrow`

`Arrow` objects are used to represent sensors, actuators, boundary conditions, ... They are characterized by indices giving their sensor set, deformation set, channel (deformation number), and arrow type. They can be modified using calls or the form (see `fecom setobject` commands)

```
cf = feplot; % get sdth object handle
cf.o(2) = 'sen 1 def 1 ch 3'
```

The *SDT* currently supports stick sensors (object type `3`) and arrows at the sensor tip (type `7`). Other arrow types will eventually be supported.

`text`

`fecom` text objects are vectorized lists of labels corresponding to nodes, elements, DOFs, ... They can be initialized using `fecom text` commands and deleted with `textoff`. You can use `cf.o(i)` (see `fecom go` commands) to get handles to the associated MATLAB text objects and thus set font name size, ... `set(cf.o(1), 'fontsize', 7)` for example.

Data arrays `feplot` stores information in various data arrays `cf.mdl` for the model, `cf.def(i)` for the definition of deformations, `cf.sel(i)` for element selections for display and `cf.sens(i)` for sensor selections.

`def`

deformation sets describe sets of deformations at a number of DOFs. Initialized using the `InitDef` command or `cf.def(i)={def,dof}`. A deformation set is characterized by fields

`.def`	one real or complex deformation per column. The `fecom ch` command allows a selection of which modes are shown.
`.DOF`	DOF definition vector giving the meaning of each row in the `.def` field (see `mdof` page 146). `feplot` currently only retains translation DOFs (DOFs `01` to `03` corresponding to u, v, w translations along global coordinate axes and DOFs `07` to `09` corresponding to $-u$, $-v$, $-w$ translations). All undeclared translations are assumed to be zero.
`.scale`	the first row gives the DOF at which the maximum response is seen, the second row the value of this response. This information is used for automated scaling (see `fecom scale` commands)
`.lab`	label associated to each deformation (see `fecom head` commands).

`mdl`

The model currently displayed is stored in `cf.mdl` `fecom initmodel`.

`sel`

element selections describe a selection of elements to be displayed. The standard selection displays all elements of all groups. `fecom sel` commands let you define selections that only display some elements.

`.selelt`	string used for element selection
`.vert0`	position of vertices (nodes) in the undeformed configuration
`.node`	node numbers associated to the various vertices
`.cna`	array (as many as currently declared deformations) of sparse observation matrices giving the linear relation between deformation DOFs and translation DOFs at the selection nodes. The observation matrix gives all x translations followed by all y translations and all z translations.
`.fs`	face definitions for true surfaces (elements that are not represented by lines or points)
`.f2`	face definitions for lines (if any)
`.f1`	face definitions for points (if any)
`.fvcs`	FaceVertexCData for true surfaces (see `fecom ColorData` commands)
`.fvc2`	FaceVertexCData for lines
`.fvc1`	FaceVertexCData for points

`sens`

sensor selections describe sets of sensors. Sensor selections are used to display the response at measurement locations through stick or arrows. Initialized using the `InitSens` command.

`.vert0`	position of vertices (nodes) in the undeformed configuration
`.node`	node numbers associated to the various vertices
`.ntag`	numerical tag identifying each sensor
`.dir`	direction associated with each sensor
`.cta`	array (as many as currently declared deformations) of sparse observation matrices giving the linear relation between deformation DOFs and measurements.
`.opt`	[Created]
`.arrow`	defines how the arrow is related to the measurement

See also `fecom`, `femesh`, `feutil`, tutorial in section 5.4

fesuper

Purpose User interface for superelement support.

Syntax

```
fesuper('CommandString')
[out,out1] = fesuper('CommandString',in1,in2)
```

Description Superelements are global variables of the general form `SE`*`Name`* with different fields allowing `fe_super` to perform the usual tasks of an element function. By default these variables are not declared as global in the base workspace. Thus to access them from there you need to use `global SE`*`Name`*.

The `fesuper` user interface provides standard access the different fields (see `fe_super` for a list of those fields). The following sections describe currently implemented commands and associated arguments (see the `commode` help for hints on how to build commands and understand the variants discussed in this help). An example of superelement use is given in the `d_cms2` demonstration.

Warnings. In the commands superelement names must be followed by a space (in most other cases user interface commands are not sensitive to spaces).

`Copy` *`Name NameNew`*

Makes a copy (duplicate) of superelement *`Name`* called *`NameNew`*. Same as

```
global SEName SENameNew; SENameNew=SEName;
```

`Get` *`Name`* `...`

Get properties from a superelement. Properties directly stored as fields of the superelement variable (see `fe_super` for details on those fields). The easiest way to access superelement fields is to declare the array as global in your current workspace `global SE`*`Name`* and access the fields directly `SE`*`Name`*`.DOF`, ...

`.DOF`	DOF definition vector for the superelement
`.Elt`	initial model description matrix
`.line`	node line for wire frame plots
`.K{i}`	superelement matrices
`.Node`	nominal nodes
`.Opt`	options
`.patch`	patch matrix for patch plots
`.ref`	coordinate transformation specification
`.tr`	reduction basis

`Info` *`Name`*

Outputs a summary of current properties of the superelement *`Name`*.

`[Load, Save]` *`FileName`*

Loading and saving superelements. `Load` *`FileName`* loads superelements (variables with name of the form `SE`*`Name`*) present in the file. `Save` *`FileName`* *`Name1`* *`Name2`* ... saves superelements *`Name1`* *`Name2`* ... in the file. Note that these commands are really equivalent to `global SE`*`Name`*`;save FileName SE`*`Name`* and `global SE`*`Name`*`;load FileName SE`*`Name`*.

`Make` *`Name`* `[generic, complete]`

`elt=fesuper('make `*`Name`*` generic')` takes a unique superelement and makes it generic (see `fe_super` for details on generic superelements). `Opt(1,1)` is set to `2`. `SE`*`Name`*`.DOF` is transformed to a generic DOF form. The output `elt` is a model description matrix for the nominal superelement (header row and one element property row). This model can by used by `femesh` to build structures that use the generic superelement several times (see the `d_cms2` demo).

`make complete` adds zero DOFs to nodes which have less than 3 translations (DOFs `.01` to `.03`) or rotations (DOFs `.04` to `.06`). Having complete superelements is important to be able to rotate them (used for generic superelements with a `Ref` property).

`New` *`Name`*

New unique superelement declaration using the general format
`fesuper ('New `*`Name`*`',FEnode,FEelt)`. If a superelement called *`Name`* exists it is

erased. The `Node` and `Elt` properties are set to those given as arguments. The `Patch` property used by `feplot` for display is initialized.

`Set` *`Name`* `...`

Set superelement properties. For `Node`, `DOF`, the calling format is `fesuper ('Set` *`Name Var`*`',Var)`. Other specific formats are

`fesuper('Set`*`Name`* `Ref 1` *`n1 n2 n3 n4 i1 i2 i3 i4`*`',node)` is used to specify the coordinate transformation property of a superelement for a type `1` coordinate transformation (the only currently available). *`n1`*... *`n4`* are node numbers (the default for `node` is `SE.Node`) for the reference basis and *`i1,i2,i3,i4`* give the positions of these node numbers in the element row of a generic superelement. You may omit *`i1,i2,i3,i4`* specification.

`fesuper('set` *`Name`* `k`*`i type`*`',Mat)` sets the superelement matrix `K{i}` to `Mat` and its type to *`type`*. The size of `Mat` must be coherent with the superelement DOF vector. *`type`* is a positive integer giving the meaning of the considered matrix (`1` stiffness, `2` mass, ...).

`fesuper('set` *`Name`* `Line',elt)` where a model description matrix `elt` rather than a connectivity matrix is given. **Starting with *SDT 4* line properties are no longer used**.

`fesuper('set` *`Name`* `mk',pl,il)` assembles the mass and stiffness matrices linked to the nominal element description matrix and the given material `pl` and element `il` property matrices. This nominal stiffness is stored as `K1` and this nominal mass as `K2`. For a reduced model assembly or a case with some boundary conditions fixed, `TR` can and should be set first.

`fesuper('set` *`Name`* `Patch',elt)` sets the patch property based on a model description matrix `elt` rather than a patch matrix.

`fesuper ('set` *`Name`* `ProID` *`i`*`')` sets `ProID` to *`i`*. `ProID` numbers must be positive integers. For unique superelements this is stored in `SE`*`Name`*`.Opt(1,2)`).

`fesuper('set` *`Name`* `TR',tr,mdof,adof)` sets the `TR` property and projects any existing matrix accordingly. An empty `mdof` can be used to fix all DOFs not in `adof` (same meaning as `adof` in `fe_mk`). If `tr` is not empty (a projection is defined) `mdof` and `adof` must be consistent.

See also `fe_super`, `upcom`, section 6.2.2, section 5.5.2

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 7.5). `fe_c` provides simplified ways to extract the indices of particular DOFs (see also section 7.10) 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 151). 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** (see b,c page 26). 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, nor2ss
Section 5.2

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`)
`FVol`	volume loads defined on elements (handled by `fe_load`)
`Info`	used to stored non standard entries
`KeepDOF`	used to eliminated DOFs not specified by the stack data
`map`	field of normals at nodes
`mpc`	multiple point constraints
`par`	are used by `upcom` to define physical parameters (see `upcom par` commands
`rigid`	linear constraints associated with rigid links
`SensDof`	(SDT) Translation sensor definitions
`SensStrain`	(SDT) Strain 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

- `AddToCase (i)` allows specification of the active case (by number in the model stack) for multiple case models. See the example below.

- `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 7.13).

- `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. The nominal calling format is `Case = fe_case(model,'gett',Case)` which fills in the `Case.T` and `Case.DOF` fields.

- `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')
```

Commands for avanced constraint generation

`Build` *Sec* `epsl` *d*

`model = fe_cyclic('build (N) epsl (d)',model,LeftNodeSelect)` is used to append a cyclic constraint entry in the current case.

`ConnectionPivot`

This commands generates a set of MPC defining a pivot connection between two sets of nodes. The command specifies the DOFs contraint at the pivot (in the example DOF 6 is free), the local z direction and the location of the pivot node. One then gives the model, the connection name, and node selections for the two sets of nodes.

```
model=demosdt('demoTwoPlate');
model=fe_caseg('Connection Pivot 12345 0 0 1 .5 .5 -3 -id 1111',model, .
 'pivot','group1','group2');
def=fe_eig(model);feplot(model,def)
```

The string `-id` *`value`* can be added to the command to specify a MPC ID for export to other software.

ConnectionSurface

`fe_caseg('Connection surface` *`DOFs`*`',model,'name',`*`NodeSel1`*`,`*`Eltsel2`*`);` generates a set of MPC connecting of *`DOFs`* of a set of nodes selected by *`NodeSel1`* (this is a string that will be passed to `feutil` as a `horzcat('GetNode',NodeSel1)` command) to a surface selected by *`EltSel2`* (this is a string that will be passed to `feutil` as a `horzcat('SelElt',EltSel2)` command). The following example links x and z translations of two plates

```
model=demosdt('demoTwoPlate');
model=fe_caseg('Connection surface 13 -id 1111',model,'pivot', ...
  'z==0', ...          % Selection of nodes to connect
  'withnode {z==.1}'); % Selection of elements for matching
def=fe_eig(model);feplot(model,def)
```

The string `-id` *`value`* can be added to the command to specify a MPC ID for export to other software.

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 `{Type, Name, Data}`. `Type` is either `'KeepDof'` or `'FixDof'`. `Name` is a string identifying the entry. `data` is a column DOF definition vector (see section 7.10) 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 a DOF specification to the `findnode` command. For example `x==0 -dof 1 2` fixes DOFs x and y on the `x==0` plane.

The following syntax is used in the final example of the section:

```
model = fe_case(model,'AddToCase 1','FixDof','clamped dofs','z==0');
```

KeepDof

`KeepDof` entries correspond to rows of the `Case.Stack` cell array giving `{Type, Name, Data}`. `Type` is either `'KeepDof'` or `'FixDof'`. `Name` is a string identifying the entry. `data` is a column DOF definition vector (see section 7.10) or a string defining a node selection command.

The following syntax is used in the final example of the section:

```
model=fe_case(model,'AddToCase1','KeepDof','3-D motion',[.01 .02 .03]');
```

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 whos 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 optionnal vector of slave DOFs in Data.DOF. This vector is currently ignored and the slave DOF is taken to be the first occurence of the value 1 on each row of c. If there is no such occurence, an error is generated.

rigid

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 following example generates the mesh of a square plate with a rigid edge

```
femesh(';testquad4;divide 10 10;addsel');
model=femesh('model');

% 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(model,'addtocase1','rigid','Rigid edge',FEel0);

% Compute and display modes
def=fe_eig(model,[6 20 1e3]);
feplot(model,def);fecom(';view3;ch8;scd.1');
```

SensDOF (SDT)

SensDOF entries are rows of the Case.Stack cell array giving { 'SensDOF', Name, data}. Name is a string identifying the entry. data is a structure with fields

.tdof	Currently .tdof should be a dof definition vector. Eventually, *SDT* will support a seven column matrix where each row describes a sensor by [SensID NodeID nx ny nz Type] giving a sensor identifier (integer or real), a node identifier (positive integer), the projection of the measurement direction on the global axes (if relevant).
.cta	is an observation matrix associated with the observation equation $\{y\} = [c]\{q\}$. This is build using the fe_case sens command as illustrated below.
.DOF	DOF definition vector for the .cta field
.lab	a cell array of string labels for each sensor defined in .tdof

```
model=demosdt('demo ubeam mix');
model=fe_case(model,'addtocase', ...
  'SensDof','Outputs',[343.01 343.02 347.03]');
Case=fe_case(model,'GetCase');
Sens = fe_case(model,'sens',Case)
Load = fe_load(model,Case)
```

SensStrain (SDT)

SensStrain entries are rows of the Case.Stack cell array giving { 'SensDOF', Name, data}. Name is a string identifying the entry. data is a structure with fields

.Node	Positions of the strain sensors. x,y,z coordinates in global coordinate system or standard node.
.dir	first direction for strain measurement. This vector need not be normalized.
.dir2	second direction for strain measurement in order to measure non-diagonal terms of the strain tensor. If undefined one measures axial strains in directions given by .dir. This vector need not be normalized.
.eltsel	optional field specifying an element selection. The faces of that selection which are contained in the nodes selected with data.sel will be loaded.
.lab	a cell array of string labels for each sensor

Strain

sensors are supported for volume and shell elements. Strains $\{d_1\}^T [\epsilon] \{d^2\}$ are computed at the matching position of the element with the nearest center of gravity.

For shells, one accounts for the offset from the neutral fibre if the sensor position is not located on the surface defined by element nodes. The approach used is only valid if rotations at the shell nodes correspond to rotations in the global coordinate system.

For volumes, reliable results require that the sensor be located within the volume defined by nodes.

Note that the observation matrix generated for these sensors is a direct estimation of strain within a single element. Since FEM solutions do not typically lead to continuous strain fields, the estimate may be quite wrong in areas of rapid strain variations where the FEM result is not properly converged.

A sample call takes the form

```
[model,def]=femesh('teststruct quad4');
data=struct('Node',[.4 .4 .01;.5 .5 .2],'dir',[1 0 0;1 1 0]);
```

```
model=fe_case(model,'SensStrain','Outputs',data);
Sens = fe_case(model,'sens')
```

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

```
model = femesh('test ubeam plot');
% creating Case1 with 3D-motion (KeepDof)
model=fe_case(model,'AddToCase1','KeepDof',...
  '3-D motion',[.01 .02 .03]');
% specifying clamped dofs (FixDof)
model = fe_case(model,'AddToCase 1','FixDof','clamped dofs','z==0');
% creating a volumic load
data  = struct('sel','GroupAll','dir',[1 0 0]);
model = fe_case(model,'AddToCase 1','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
cf.def=def;
fecom(';undef;triax;showpatch;promodelinit');
```

See also `fe_mk`, `fe_case`

fe_ceig

Purpose Computation and normalization of complex modes associated to a second order viscously damped model.

Syntax

```
[psi,lambda] = fe_ceig( ... )
lambda       = fe_ceig(m,c,k)
def          = fe_ceig( ... )
         ... = fe_ceig(m,c,k)
         ... = fe_ceig({m,c,k,mdof},eigopt)
         ... = fe_ceig({m,c,k,T,mdof},eigopt)
         ... = fe_ceig(model,eigopt)
```

Description Complex modes are solution of the second order eigenvalue problem (see section 2.5 for details)

$$[M]_{N\times N}\{\psi_j\}_{N\times 1}\lambda_j^2+[C]\{\psi_j\}\lambda_j+[K]\{\psi_j\}=0$$

where modeshapes `psi`$=\psi$ and poles $\Lambda=\left[\backslash\lambda_{j\backslash}\right]$ are also solution of the first order eigenvalue problem (used in `fe_ceig`)

$$\begin{bmatrix} C & M \\ M & 0 \end{bmatrix}_{2N\times 2N}\begin{bmatrix} \psi \\ \psi\Lambda \end{bmatrix}_{2N\times 2N}[\Lambda]_{2N\times 2N}+\begin{bmatrix} K & 0 \\ 0 & -M \end{bmatrix}\begin{bmatrix} \psi \\ \psi\Lambda \end{bmatrix}=[0]_{2N\times 2N}$$

and verify the two orthogonality conditions

$$\psi^TC\psi+\Lambda\psi^TM\psi+\psi^TM\psi\Lambda=I \quad \text{and} \quad \psi^TK\psi-\Lambda\psi^TM\psi\Lambda=-\Lambda$$

`[psi,lambda] = fe_ceig(m,c,k)` is the old low level call to compute all complex modes. For partial solution you should use `def = fe_ceig(model,ceigopt)` where model can be replaced by a cell array with `{m,c,k,mdof}` or `{m,c,k,T,mdof}` (see the example below). Using the projection matrix `T` generated with `fe_case('gett')` is the proper method to handle boundary conditions.

Options give `[CeigMethod EigOpt]` where `CeigMethod` can be 0 (full matrices), 1 (real modes then complex ones on the same basis) 2 and 3 are refined solvers available with the VISCO extension. `EigOpt` are standard `fe_eig` options.

Here is a simple example of `fe_ceig` calls.

```
model=femesh('testubeam');model.DOF=[];
```

```
model=fe_case(model,'fixdof','Base','z==0');
[m,k,model.DOF]=fe_mk(model,'options',[0 2]);
Case=fe_case(model,'gett');

kc=k*(1+i*.002); % with hysteretic damping
def1=fe_ceig({m,[],kc,model.DOF},[1 6 10 1e3]);       % free modes
def2=fe_ceig({m,[],kc,Case.T,Case.DOF},[1 6 10 1e3]); % fixed modes
```

See also fe_eig, fe_mk, nor2ss, nor2xf, section 2.3 section **??**

fe_coor

Purpose Coordinate transformation matrices for Component Mode Synthesis problems.

Syntax

```
[t] = fe_coor(cp)
[t,nc] = fe_coor(cp,opt)
```

Description The different uses of `fe_coor` are selected by the use of options given in the argument `opt` which contains `[type method]` (with the default values `[1 3]`).

`type=1` (default) the output `t` is a basis for the kernel of the constraints `cp`

$$\text{range}([T]_{N\times(N-NC)}) = \ker([c]_{NS\times N})$$

$NC \leq NS$ is the number of independent constraints.

`type=2` the output argument `t` gives a basis of vectors linked to unit outputs followed by a basis for the kernel

$$T = \left[[T_U]_{N\times NS} \, [T_K]_{N\times(N-NS)} \right] \text{ with } [c]_{NS\times N} \, [T] = \left[\left[{}^{\backslash} I_{\backslash} \right] [0]_{NS\times(N-NS)} \right]$$

If $NC < NS$ such a matrix cannot be constructed and an error occurs.

`method` the kernel can be computed using : `1` a singular value decomposition `svd` (default) or `3` a `lu` decomposition. The `lu` has lowest computational cost. The `svd` is most robust to numerical conditioning problems.

Usage `fe_coor` is used to solve problems of the general form

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

which are often found in CMS problems (see section 6.1.6 and [36]).

To eliminate the constraint, one determines a basis T for the kernel of $[c_{int}]$ and projects the model

$$\begin{array}{c} \left[T^T M T s^2 + T^T C T s + T^T K T\right] \{q_R(s)\} = \left[T^T b\right] \{u(s)\} \\ \{y(s)\} = [cT] \{q_R(s)\} \end{array}$$

See also Section 7.13, `fe_c`, the `d_cms` demo

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 7.9. 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

`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.

getcurve

`curve=fe_curve('getcurve',model,curve_name);`
extracts curve `curve_name` from the `.Stack` field of `model`.

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`. *`input_channels`* are input channels numbers in `frames`. 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.

N.B.: 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);`

```
out=fe_curve('testframe');       % 3 DOF system response
frames{1}.X=out{1}.X;            % build frame as a cell array,
frames{1}.Y=[out{1}.Y out{2}.Y]; %  even for single frame
% Time vector in .X field, measurements in .Y field
% Noise signal on first input, response on second input
frf=fe_curve('h1h2 1',frames); % compute FRF
figure(1); semilogy(frf.X,abs(frf.H1),'b'); hold on;
semilogy(frf.X(1:5:end),abs(frf.H2(1:5:end)),'.r');
legend('H1','H2');
xlabel('Freq. - Hz'); ylabel('Frequ. Response Function - m/N');
```

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 interest for the response spectra.

```
pw0=fileparts(which('gartfe'));
st=sprintf('read %s',fullfile(pw0,'bagnol_ns.cyt'));
bagnol_ns=fe_curve(st); % read the acceleration time signal

bagnol_ns.yunit=fe_curve('datatype','Acceleration');
st=sprintf('read %s',fullfile(pw0,'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

```
y = fe_curve('returny',model,curve_name,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,[FRF ; zeros(length(FRF),1)].');

% 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');
```

`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)`.

`test[Ramp,Ricker]`

`out=fe_curve('TestRamp NStep FinalValue')` generates a ramp composed of `NStep` steps from 0 to `FinalValue`.

`out=fe_curve('TestRicker Duration Nstep Amplitude TotalTime')` generates Ricker functions representing impacts. For example:

```
C1=fe_curve('test ramp 20 2');
C2=fe_curve('TestRicker .6 120 2 1.2');
figure(1);plot(C1.X,C1.Y,'-',C2.X,C2.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.

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_cyclic

Purpose Support for cyclic symmetry computations.

Syntax

```
model=fe_cyclic('build NSEC',model,LeftNodeSelect)
fe_cyclic('display NSEC',model,def,EltSelect)
def=fe_cyclic('eig NDIAM',model,EigOpt)
```

Description `fe_cyclic` groups all commands needed to compute responses assuming cyclic symmetry. For more details on the associated theory you can refer to [43].

Assemble

This command supports the computations linked to the assembly of gyroscopic coupling, gyroscopic stiffness and tangent stiffness in geometrically non-linear elasticity. The input arguments are the model and the rotation vector (in rad/s).

```
model=demosdt('demo sector all');
[m,k,mdof]=fe_mknl(model);
[c_g,k_g,k_e]=fe_cyclic('assemble',model,[0 0 1000]); %
def=fe_eig({m,k,mdof},[6 20 0]);    % Non rotating modes
def2=fe_eig({m,k_e,mdof},[6 20 0]); % Rotating mode shapes
[def.data def2.data]
```

Note that this command does not YET support cyclic symmetry conditions so that it must run on the full disk.

Build

`model=fe_cyclic('build` *`nsec`*`',model,'LeftNodeSelect')` adds a `cyclic` symmetry entry in the model case. It automatically rotates the nodes selected with `LeftNodeSelect` by 2π/nsec and finds the corresponding nodes on the other sector face. The default for `LeftNodeSelect` is `'GroupAll'` which selects all nodes.

Display

`fe_cyclic('display` *`nsec`*`',model,def,'EltSelect')` repeats the model elements selected with `EltSelect` (default is `'GroupAll'`) and displays the resulting model and deformations `def` on `nsec` sectors.

Eig

`def=fe_cyclic('eig ndiam',model,EigOpt)` computes `ndiam` diameter modes using the cyclic symmetry assumption. For `ndiam>0` these modes are complex to account for the inter-sector phase shifts. `EigOpt` are standard options passed to `fe_eig`.

This example computes the two diameter modes of a three bladed disk also used in the `d_cms2` demo.

```
model=demosdt('demo sector');
model=fe_cyclic('build 3',model,'groupall');
fe_case(model,'info')
def=fe_cyclic('eig 2',model,[6 20 0 11]);
fe_cyclic('display 3',model,def)
```

TEig [,Nast,ReadNast]

The `TEig` commands seek to compute modes for multiple diameters in a single job. With a model saved in structure `Up`, you can specify target diameters, eigenvalue options and possibly a file name to save intermediate results

```
model=demosdt('demo sector 5');
fe_case(model,'info')
def=fe_cyclic('teig 0 1 3 5',model,[6 20 0 11],tempname);
fe_cyclic('display 5',model,def)
```

For `FEMLink`users, the `TEigNast` command lets you use NASTRAN to generate the same solution.

```
def=fe_cyclic('teignast 0 1 2 3',Up,'NastranBulkName.bdf');
```

Finally `def=fe_cyclic('teig',[0 1 3],d0,d1,d3);` can be used to merge modes computed elsewhere in the typical multi-diameter solution.

Reduce

The `Reduce` command is used to generate a disk model from a set of modes associated multiple diameters symmetry. The general call is

```
DISK=fe_cyclic('reduce',model,def);
```

See also Section 7.13

fe_eig

Purpose Computation of normal modes associated to a second order undamped model.

Syntax

```
DEF = fe_eig(model,opt)
DEF = fe_eig({m,k,mdof},opt)
DEF = fe_eig({m,k,T,mdof},opt)
[phi, wj] = fe_eig(m,k)
[phi, wj, kd] = fe_eig(m,k,opt,imode)
```

Description The normal modeshapes `phi`$=\phi$ and frequencies `wj= sqrt(diag(`Ω^2`))` are solution of the undamped eigenvalue problem (see section 2.2)

$$-[M]\{\phi_j\}\omega_j^2+[K]\{\phi_j\}=\{0\}$$

and verify the two orthogonality conditions

$$[\phi]^T[M]_{N\times N}[\phi]_{N\times N}=[I]_{N\times N} \text{ and } [\phi]^T[K][\phi]=\left[{}^{\backslash}\Omega_{j}^{2}{}_{\backslash}\right]$$

The outputs are the data structure `DEF` (which is more appropriate for use with high level functions `feplot`, `nor2ss`, ... since it keeps track of the signification of its content, frequencies in `DEF.data` are then in **Hz**) or the modeshapes (columns of `phi`) and frequencies `wj` in **rad/s**. Note how you provide `{m,k,mdof}` in a cell array to obtain a `DEF` structure without having a model.

The optional output `kd` corresponds to the factored stiffness matrix. It should be used with methods that do not renumber DOFs.

`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` **2 default** full matrix solution. Cannot be used for large models.
6 IRA/Sorensen solver (preferred partial solver).
5 Lanczos solver allows specification of frequency band rather than number of modes. To turn of convergence check add `2000` to the option (`2105`, `2005`, ...).

106, 104 same as the corresponding methods but no initial DOF renumbering. This is useless with the default `ofact('methodspfmex')` which renumbers at factorization time.

Obsolete methods

0 SVD based full matrix solution

1 subspace iteration which allows to compute the lowest modes of a large problem where sparse mass and stiffness matrices are used.

3 Same as **5** but using `ofact('methodlu')`. **4** Same as **5** but using `ofact('methodchol')`.

`nm`	number of modes to be returned. A non-integer or negative `nm`, is used as the desired `fmax` in **Hz** for iterative solvers (method 5 only).
`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 and Lanczos methods)

Finally, a set of vectors `imode` can be used as an initial guess for the subspace iteration method (`method 1`).

Notes

- The default full matrix algorithm (`method=2`) cleans results of the Matlab `eig` function. Computed modes are mass normalized and complex parts, which are known to be spurious for symmetric eigenvalue problems considered here, are eliminated. The alternate algorithm for full matrices (`method=0`) uses a singular value decomposition to make sure that all frequencies are real. The results are thus wrong, if the matrices are not symmetric and positive definite (semi-positive definite for the stiffness matrix).
- The preferred partial solver is `method 6` which calls `eigs` (ARPACK) properly and cleans up results.
- The subspace iteration and Lanczos algorithms are rather free interpretation of the standard algorithms (see Ref. [33] for example).
- The Lanczos algorithm (methods `3,4,5`) is much faster than the subspace iteration algorithm (method `1`). A double orthogonalization scheme and double restart usually detects multiple modes.
- 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 occasionally need to play around a little). If you do not find the expected number of rigid body modes, this is can be reason.

Example

Here is an example containing a high level call

```
model =demosdt('demo gartfe');
cf=feplot;cf.model=model;
cf.def=fe_eig(model,[6 20 1e3 11]);
fecom chc10
```

and the same example with low level commands

```
model =demosdt('demo gartfe');
[m,k,mdof] = fe_mknl(model);
cf=feplot;cf.model=model;
cf.def=fe_eig({m,k,mdof},[5 20 1e3]);fecom chc10
```

See also fe_ceig, fe_mk, nor2ss, nor2xf

fe_exp

Purpose Expansion of experimental modeshapes.

Syntax

```
emode = fe_exp(yTest,c,T)
emode = fe_exp(yTest,sens,T,opt)
yExp  = fe_exp(yTest,fTest,sens,m,k,mdof,freq,opt)
```

Description A unified perspective on interpolation and the more advanced finite element based expansion methods are discussed in the tutorial 4.3. An example is treated in detail in the `gartco` demonstration. This section gives a list of available methods with a short discussion of associated trade-offs.

`Interpolation`

Interpolation methods can be implemented easily by constructing a basis `t` of possible displacements and minimizing the test error as discussed in section 4.3.2.

For example, if node 2 is placed at a quarter of the distance between nodes 1 and 3 whose motion is observed. A linear interpolation for translations in the x direction is built using

```
ndof = [1.01;2.01;3.01]; T= [1 0;3/4 1/4;0 1];
yExp = fe_exp(yTest,fe_c(ndof,[1.01;3.01])*T,T)
```

For expansion of this form, `T` must contain at most as many vectors as there are sensors. In other cases, a solution is still returned but its physical significance is dubious.

`Modal, Serep`

Modal or SEREP expansion is a subspace based expansion using the subspace spanned by low frequency target modes. With a sensor configuration defined (`sens` defined using `fe_sens`), a typical call would be

```
[phi,wj] = fe_eig(m,k,[105 50 1e3]);
TargetModes = phi(:,[IndicesOfTargetModes]);
mdex = fe_exp(IIres.',sens,TargetModes);
```

This method is very easy to implement. Target modes can be imported from an external code. A major limitation is the fact that results tend to be sensitive to target mode selection.

You can impose that an orthogonal linear combination of the modes is used for the expansion using `mdex = fe_exp(IIres.',sens,TargetModes,2);`. This is motivated for cases where both test and analysis modeshapes are mass normalized and will provide mass orthonormal expanded modeshapes [44]. In practice it is rare that test results are accurately mass normalized and the approach is only implemented for completeness.

Static

Static expansion can be obtained using two different approaches. If constraint modes (or attachment modes for structures without rigid body modes) are imported (or computed using `[T] = fe_reduc('static',m,k,mdof,sdof)`), static expansion can be considered as a subspace method and the solution is found with

```
mdex = fe_exp(IIres.',sens,T);
```

The subspace can also be computed by `fe_exp`, using

```
mdex = fe_exp(IIres.',0,sens,m,k,mdof);
```

which will handle arbitrary sensor configurations defined in `sens`.

The main limitation with static expansion is the existence of a frequency limit (first frequency found when all sensors are fixed). `[mdex,phi_fixed] = fe_exp(IIres.', 0, sens, m,k,mdof);` returns an estimate of the first 10 fixed sensor modes. If the first frequency is close to your test bandwidth, you should consider using dynamic expansion.

Dynamic, RBDE

Dynamic expansion is supported at a single frequency or at one frequency for each deformation to be expanded using

```
mdex = fe_exp(yTest,fExp*2*pi,sens,m,k,mdof);
```

Reduced basis dynamic expansion (RBDE) is supported using

```
mdex = fe_exp(yTest,fExp*2*pi,sens,m,k,mdof,T);
```

where `T` typically contains normal and attachment modes (see `gartco` for an example). Note that, when using reduced bases, you can provide `T'*m*T` and `T'*k*T` instead of `m` and `k` which saves a lot of time if these matrices have already been computed.

MDRE, MDRE-WE

Minimum dynamic residual expansion (MDRE) is currently only implemented for normal modeshape expansion. Furthermore, computational times are generally only acceptable for the reduced basis form of the algorithm. A typical call would be

```
mdex = fe_exp(yTest,fExp*2*pi,sens,m,k,mdof,T,'mdre');
```

where T contains normal and attachment modes but often needs to be renormalized using T = fe_norm(T,m,k) to prevent numerical conditioning problems (see gartco for an example). Note that, when using reduced bases, you can provide T'*m*T and T'*k*T instead of m and k which saves a lot of time if these matrices have already been computed.

MDRE-WE (Minimum dynamic residual expansion with measurement error) iteratively adjusts the relative weighting γ_j between model and test error in (4.10). Input arguments specify a starting value for γ_j and a relative error bound. The initial value for γ_j is increased (multiplied by 2) until $\epsilon_j / \|\{y_{jTest}\}\|$ is below the given bound. A typical call would be $\epsilon_j / \|\{y_{jTest}\}\|$

```
opt=struct('type','mdrewe','gamma',1,'MeasErr',.1);
yTest=IIres.';wTest=IIpo(:,1)*2*pi;
[mdex,opt,mdexr,err]=fe_exp(yTest,wTest,sens,m,k,mdof,T,opt);
```

where the opt in the output gives the adjusted values of γ_j, mdexr is the expanded vector expressed in the generalized coordinates associated with T, and err gives the objective function value (first row) and relative error (second row).

See also fe_sens, fe_reduc, section 4.3, gartco demo.

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,'AddToCase 1','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 and prescribed displacements applied to DOFs. Type specific `data` is a structure with fields

`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`)
`data.lab`	can associated a label to each load (column of `data.def`

```
model = femesh('testubeam plot');
r1=struct('DOF',365.03,'def',1.1); % 1.1 N at node 365 direction z
Case1=fe_case('DofLoad','PointLoad',r1, ...
  'DofLoad','Short Format',362.01); % Short format for unit loads
Load = fe_load(model,Case1);
cf.def=Load;fecom(';scaleone;undefline;ch1 2') % display
```

`FVol`

`data` is a structure with fields

`data.sel`	model description matrix or element selection
`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 .

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

```
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);
```

Note that `feutil('mode2dof')` provides translation to this format from a function defined at nodes.

FSurf

`data` is 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. Alternatively, `data.sel` can contain any valid node selection (using string or cell array format).
`data.set`	Alternative specification of the loaded face by specifying a `set` name to be found in `model.Stack`
`data.eltsel`	optional field specifying an element selection. The faces of that selection which are contained in the nodes selected with `data.sel` will be loaded.
`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.

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

```
model = femesh('testubeam plot');
data=struct('sel','x==-.5', ...
   'eltsel','withnode {z>1.25}','def',1,'DOF',.19);
Case1=struct('Stack',{{'Fsurf','Surface load',data}});
Load = fe_load(model,Case1); cf.def=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};
Case1=struct('Stack',{{'Fsurf','Surface load',data}});
Load = fe_load(model,Case1); cf.def=Load;
```

Alternatively, one can specify the surface by refering to a `set` entry in `model.Stack`, as shown in the following example

```
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);cf.def=Load;
```

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 5.1.4). For general information about material properties, you should refer to section 7.3. For information about element properties, you should refer to section 7.4.

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('unit')` 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.

`Get[pl,il]`

`pl = fe_mat('getpl',model)` is used to robustly return the material property matrix `pl` (see section 7.3) 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 7.3).

`[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 8

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 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,'DO` 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

```
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 7.13.

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 7.13), Case.GroupInfo fields (detailed below) and Case.MatGraph (preallocated sparse matrix associated with the model topology for optimized (re)assembl
Case.GroupInfo is a cell array with rows giving information about each element group in the model. The meaning of the columns is as follows

DofPos	int32 matrix whos 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 whos columns describe information each element of the group. Pointers has one column per element giving [OutSize1 OutSize2 u3 NdNRule MatDes IntegOffset ConstitOffset StateOffset]
Integ	int32 matrix storing integer values used to describe the element formulation of the group
Constit	double matrix storing integer values used to describe the element formulation of the group
gstate	double matrix whos columns describe the internal state of each element of the group. This is used to store local bases in linear plate/shell elements, and stress states in non-linear elements with internal variables.
ElMap	int32 element map matrix.
InfoAtNode	struct used to store internal variables that are defined at nodes rather than elements (integration points). This is not currently used by any element.
EltConst	struct used to store element formulation information (integration rule, constitutive matrix topology, etc.) Details on this data structure are given under integrules.

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).

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

```
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 and stiffness are supported in `fe_cyclic`;
- `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 8, `fe_c`, `feplot`, `fe_eig`, `upcom`, `fe_mat`, `femesh`, etc.

fe_norm

Purpose Mass-normalization and stiffness orthonormalization of a set of vectors.

Syntax

```
To = fe_norm(T,m)
[rmode,wr] = fe_norm(T,m,k,NoCommentFlag)
[rmode,wr] = fe_norm(T,m,k,tol)
```

Description With just the mass `m` (`k` not given or empty), `fe_norm` orthonormalizes the `T` matrix with respect to the mass `m` using a preconditioned Cholesky decomposition. The result `To` spans the same vector space than `T` but verifies the orthonormality condition

$$[To]^T [M]_{N\times N} [To]_{N\times NM} = [I]_{NM\times NM}$$

If some vectors of the basis `T` are collinear, these are eliminated. This elimination is a helpful feature of `fe_norm`.

When both the mass and stiffness matrices are specified a reanalysis of the reduced problem is performed (eigenvalue structure of model projected on the basis `T`). The resulting reduced modes `rmode` not only verify the mass-orthogonality condition, but also the stiffness orthogonality condition (where $\left[\backslash \Omega_{j\backslash}^2\right]$ `=diag(wr.^2)`)

$$[\phi]^T [K] [\phi] = \left[\backslash \Omega_{j\backslash}^2\right]_{NM\times NM}$$

The verification of the two orthogonality conditions is not a sufficient condition for the vectors `rmode` to be the modes of the model. Only if $NM = N$ is this guaranteed. In other cases, `rmode` are just the best approximations of modes in the range of T.

When the fourth argument `NoCommentFlag` is a string, no warning is given if some modes are eliminated.

When a tolerance is given, frequencies below the tolerance are truncated. The default tolerance (value given when `tol=0`) is product of `eps` by the number of modes by the smallest of `1e3` and the mean of the first seven frequencies (in order to incorporate at least one flexible frequency in cases with rigid body modes). This truncation helps prevent poor numerical conditioning from reduced models with a dynamic range superior to numerical precision.

See also `fe_reduc`, `fe_eig`

fe_reduc

Purpose Utilities for finite element model reduction.

Syntax `[T,rdof,rb] = fe_reduc('command',m,k,mdof,b,rdof)`

Description `fe_reduc` provides standard ways of creating and handling bases (rectangular matrix `T`) of real vectors used for model reduction (see details in section 6.1). Input arguments are

`m`	mass matrix (can be empty for commands that do not use mass)
`k`	stiffness matrix and
`mdof`	associated DOF definition vector describing DOFs in `m` and `k`. When using a model with constraints, you can use `mdof=fe_case(model,'gettdof')`.
`b`	input shape matrix describing unit loads of interest. Must be coherent with `mdof`.
`bdof`	alternate load description by a set of DOFs (`bdof` and `mdof` must have different length)
`rdof`	contains definitions for a set of DOFs forming an iso-static constraint (see details below). When `rdof` is not given, it is determined through an LU decomposition done before the usual factorization of the stiffness. This operation takes time but may be useful with certain elements for which geometric and numeric rigid body modes don't coincide.

Accepted `fe_reduc` commands (see the `commode` help for hints on how to build commands and understand the variants discussed in this help) are

`CraigBampton` *`NM Shift Ouput`*

`[T,sdof,f,mr,kr]=fe_reduc('CraigBampton` *`NM Shift Output`*`',m,k,mdof,idof);` computes the Craig-Bampton reduction basis (6.14) associated with interface DOFs `idof`. This basis is a combination of constraint modes and fixed interface modes (whose frequencies are returned in `f`). The fixed interface modes are obtained using `fe_eig` method `6` (IRA/Sorensen).

Note that using `NM=0` corresponds to static or Guyan condensation.

`dynamic` *`w`*

`[T,rbdof,rb]=fe_reduc('dynamic` *`freq`*`', ...)` computes the dynamic response at frequency *w* to loads `b`. This is really the same as doing `(-w^2*m+k)\b` but can be significantly faster and is more robust.

flex [,nr]

`[T,rbdof,rb]=fe_reduc('flex', ...)` computes the static response of flexible modes to load `b` (which can be given as `bdof`)

$$\left[K_{Flex}^{-1}\right][b]=\sum_{j=NR+1}^{N}\frac{\{\phi_j\}\{\phi_j\}^T}{\omega_j^2}$$

where NR is the number of rigid body modes. These responses are also called static flexible responses or **attachment modes** (when forces are applied at interface DOFs in CMS problems).

The flexible response is computed in three steps:

- Determine the flexible load associated to b that does not excite the rigid body modes $b_{Flex}=([I]-[M\phi_R]\left[\phi_R^T M\phi_R\right]^{-1}[\phi_R]^T)[b]$
- Compute the static response of an isostatically constrained model to this load
$$[q_{Iso}]=\left[\begin{array}{cc}0 & 0\\ 0 & K_{Iso}^{-1}\end{array}\right][b_{Flex}]$$
- Orthogonalize the result with respect to rigid body modes
$q_{Flex}=([I]-[\phi_R]\left[\phi_R^T M\phi_R\right]^{-1}\left[\phi_R^T M\right])[q_{Iso}]$

where it clearly appears that the knowledge of rigid body modes and of an isostatic constraint is required, while the knowledge of all flexible modes is not (see [33] for more details).

By definition, the set of degrees of freedom R (with other DOFs noted Iso) forms an isostatic constraint if the vectors found by

$$[\phi_R]=\left[\begin{array}{c}\phi_{RR}\\ \phi_{IsoR}\end{array}\right]=\left[\begin{array}{c}I\\ -K_{Iso}^{-1}K_{IsoR}\end{array}\right]$$

span the full range of rigid body modes (kernel of the stiffness matrix). In other words, displacements imposed on the DOFs of an isostatic constraint lead to a unique response with no strain energy (the imposed displacement can be accommodated with a unique rigid body motion).

If no isostatic constraint DOFs `rdof` are given as an input argument, a `lu` decomposition of `k` is used to find them. `rdof` and rigid body modes `rb` are always returned as additional output arguments.

The command `flexnr` can be used for faster computations in cases with no rigid body modes. The static flexible response is then equal to the static response and `fe_reduc` provides an optimized equivalent to the MATLAB command `k\b`.

`rb`

`[rb,rbdof]=fe_reduc('rb',m,k,mdof,rbdof)` determines rigid body modes (rigid body modes span the null space of the stiffness matrix). The DOFs `rbdof` should form an isostatic constraint (see the `flex` command above). If `rbdof` is not given as an input, an `LU` decomposition of `k` is used to determine a proper choice.

If a mass is given (otherwise use an empty `[ ]` mass argument), computed rigid body modes are mass orthonormalized ($\phi_R^T M \phi_R = I$). Rigid body modes with no mass are then assumed to be computational modes and are removed.

`static [,struct]`

`[T,tdof]=fe_reduc('static',m,k,mdof,bdof)` computes the static responses to unit imposed displacements on the DOFs given in `bdof`. The output argument `tdof` is a version with no wild cards of the input argument `bdof`. If the DOFs in `bdof` are indexed I and the other C, the static responses to unit displacements are given by

$$[T] = \left[\begin{array}{c} T_I \\ T_C \end{array} \right] = \left[\begin{array}{c} I \\ -K_{CC}^{-1} K_{CI} \end{array} \right]$$

The projection of a model on the basis of these shapes is known as **static** or **Guyan condensation**. In the Component Mode Synthesis literature, the static responses to unit deformations of interface DOFs are called **constraint modes**.

Note that you may get an error if the DOFs in `bdof` do not constrain rigid body motion so that K_{CC} is singular.

`SE=fe_reduc('static struct', ... )` returns the guyan condensation as a unique superelement `SE` with the reduction basis in `SE.TR`.

See also `fe2ss`, `fe_eig`, section 6.1

fe_sens

Purpose Utilities for sensor/shaker placement and sensor/DOF correlation.

Syntax Command dependent syntax. See sections on placement and correlation below.

Placement In cases where an analytical model of the structure is available before a modal test, you can use it for test preparation, see section 4.1.3 and the associated `gartsens` demo. `fe_sens` provides sensor/shaker placement methods.

indep

```
sdof = fe_sens('indep',cphi,cdof)
```

For a given set of modes `mode` (associated to the DOF definition vector `mdof`) and possible sensor locations (DOFs described by the DOF definition vector `cdof`), the modal output matrix `cphi` is constructed using

```
cphi = fe_c(mdof,cdof)*mode
```

`sdof=fe_sens('indep',cphi,cdof)` uses the effective independence algorithm [19] to sort the selected sensors in terms of their ability to distinguish the shapes of the considered modes. The output `sdof` is the DOF definition vector `cdof` sorted according to this algorithm (the first elements give the best locations). See example in the `gartsens` demo.

mseq

`sdof = fe_sens('mseq Nsens target',DEF,sdof0)` places *Nsens* sensors, with an optional initial set *sdof0*. The maximum response sequence algorithm used here can only place meaningfully NM (number of modes in DEF) sensors, for additional sensors, the algorithm tries to minimize the off-diagonal auto-MAC terms in modes in `DEF.def` whose indices are selected by *target*. See example in the `gartsens` demo.

[ma,mmif]

```
[sdof,load] = fe_sens('ma val',po,cphi,IndB,IndPo,Ind0)
```

Shaker placement based on most important components for force appropriation of a mode. The input arguments are poles `po`, modal output shape matrix `cphi`, indices

IndB of sensor positions where a collocated force could be applied, IndPo tells which mode is to be appropriated with the selected force pattern. Ind0 can optionally be used to specify shakers that must be included.

sdof(:,1) sorts the indices IndB of positions where a force can be applied by order of importance. sdof(:,2) gives the associated MMIF. load gives the positions and forces needed to have a MMIF below the value *val* (default 0.01). The value is used as a threshold to stop the algorithm early.

ma uses a sequential building algorithm (add one position a time) while mmif uses a decimation strategy (remove one position at a time).

Correlation fe_sens provides a user interface that helps obtaining test/analysis correlation for industrial models. For a tutorial see section 4.1.

The information is stored in a structure sens with the following fields

sens.Node	node matrix for a model containing both test and FEM nodes
sens.Elt	element description matrix for the FEM model and test wire-frame display. fe_sens near, rigid, ... commands add test/FEM node and rotation interpolation links to this model to allow rebuilding of the observation matrix cta.
sens.tdof	test sensor definition which can be a DOF definition vector (see mdof page 146) or more generally a 5 column matrix with rows containing [SensID NodeID nx ny nz] giving a sensor identifier (integer or real), a node identifier (positive integer), the projection of the measurement direction on the global axes. See section 4.1.1 command for local coordinate system handling.
sens.DOF	DOF definition vector for the analysis (finite element model).
sens.cta	observation matrix for sensor motion (tdof) based on full order motion adof
sens.bas	Coordinate system definitions for nodes. In particular you may want to define a coordinate system for test nodes using the basis command.

Commands supported by fe_sens are

basis

sens= fe_sens('BasisEstimate',sens) estimates a local coordinate system for test nodes that matches the FEM model reasonably and displays the result in a fashion that lets you edit the estimated basis

```
sens=demosdt('demo gartte basis');
sens=fe_sens('basis estimate',sens)
sens=fe_sens('basis',sens, ...
  'x',     [0 1 0], ... % x_test in FEM coordinates
  'y',     [0 0 1], ... % y_test in FEM coordinates
  'origin',[-1 0.0 0.0],... % test origin in FEM coordinates
  'scale', [0.01]); % test/FEM length unit change
```

cta

`cta = fe_sens('cta',sens)` uses links defined in `sens.Elt` to build the observation matrix of test DOF motion from active FEM DOFs defined by `sens.DOF`.

info

`fe_sens('info',sens)` returns a summary of sensor configuration information currently stored in `sens`.

tdof

`tdof = fe_sens('tdof',sens.tdof)` returns the 5 column form of `tdof` if `sens.tdof` is defined as a DOF definition vector.

Note that sensors defined using a `sens.tdof` DOF definition vector use the response coordinate system information given in column 3 of `sens.Node` while the 5 column format gives sensor directions in the global FEM coordinate system. In the example above, position and displacement coordinate systems for test nodes are set to 100. Thus the sensor 1011y (`sens.tdof(1)`) is a measurement in FEM direction z.

plotlinks

`fe_sens('plotlinks',sens)` generates a standard plot showing the FEM as a gray mesh, the test wire-frame as a red mesh, test/FEM node links as green lines with end circles, and rotation interpolation links as blue lines with cross markers.

laser

`sdof = fe_sens('laser` *`px py pz`*`',sens,SightNodes)` defines sensors in a form acceptable for inclusion in `sens.tdof` based on line of sight direction from the laser scanner position *`px py pz`* to the measurement nodes *`SightNodes`*. Sighted nodes can be specified as a standard node matrix or using a node selection command such as `'NodeId>1000 & NodeId<1100'`.

`near,rigid,arigid`

Calls of the form

```
sens=fe_sens('arigid',sens,'TestNodeSelectors','FEMNodeSelectors');
```

are used to create observation matrices for sensors. Please read section 4.1.2 for more details.

`stick`

The `stick` command can be used to find an orthonormal projection of the test nodes onto the nearest FEM surface. The projected nodes are found in the `match.StickNode` field.

```
[sens,def]=demosdt('demo gartte cor');
match=fe_sens('stick',sens,'selface');
```

`wireexp`

`def = fe_sens('wireexp',sens)` uses the wire-frame topology define in `sens` to create an interpolation for un-measured directions. The following example applies this method for the GARTEUR example. You can note that the in-plane bending mode (mode 8) is clearly interpolated with this approach.

```
[sens,test_mode]=demosdt('demo gartte wire');
exp=fe_sens('wireexp show',sens);
pause
cf=feplot;cf.model=sens;
cf.def(1)=test_mode;
cf.def(2)={test_mode,exp};
fecom(';show2def;scaleequal;ch8');
legend(cf.o(1:2),'Nominal','Wire-exp')
```

By default each segment of the wire-frame is represented as a beam with a diameter chosen based on the mean inter node distance. You can specify the beam diameter in the command `def = fe_sens('wireexp diam 5e-3',sens)`.

`femesh`, `fe_exp`, `fe_c`,`ii_mac`, `ii_comac`

fe_simul

Purpose High level access to standard solvers.

Syntax

```
[Result,model] = fe_simul('Command',MODEL,CASE,OPT)
```

Description fe_simul is the generic function to compute various types of response. It allows an easy access to specialized functions to compute static, modal (see fe_eig) and transient (see fe_time) response. A tutorial may be found in section 5.3.

Once you have defined a FEM model (section 5.1), material and elements properties (section 5.1.4), loads and boundary conditions (section 5.2), calling fe_simul assembles the model (if necessary) and computes the response using the dedicated algorithm.

Note that you may access to the fe_simul commands graphically with the simulate tab of the feplot GUI. See tutorial (section 5.3) on how to compute a response.

Input arguments are :

- MODEL can be specified by four input arguments Node, Elt, pl and il (see section 5.1)
- CASE are information needed to build the load and boundary conditions is given in the .Stack (see section 5.2)
- OPT is an option vector used for some solutions.

Accepted commands are

- Static: computes the static response to loads defined in the Case. no options are available for this command

  ```
  model = demosdt('demo ubeam');cf=feplot;cf.model=model;
  data  = struct('sel','GroupAll','dir',[1 0 0]);
  model = fe_case(model,'AddToCase 1','FVol','Volumic load',data);
  [cf.def,model]=fe_simul('static',model,'Case 1');
  ```

- Mode : computes normal modes, fe_eig options can be given in the command string or as an additional argument. For modal computations, opt=[method nm Shift Print Thres] (it is the same vector option as for fe_eig). This an example to compute the first 10 modes of a 3D beam :

```
model = demosdt('demo ubeam');cf=feplot;cf.model=model;
[cf.def,model]=fe_simul('mode',model,'Case 1',[6 10]);
```

- `Time` : computes the time response. You must specify which algorithm is used (`Newmark`, Discontinuous Galerkin `dg` or `Newton`)). For transient computations, `opt= [beta alpha t0 deltaT Nstep Nf]` (it is the same vector option as for `fe_time`). Calling time response with `fe_simul` does not allow initial condition. This is an example of a 1D bar submitted to a step input :

```
model=demosdt('demo bar');
[def,model]=fe_simul('time newmark',model, ...
              'Case 1',[.25 .5 0 1e-4 50 10]);
def.DOF=def.DOF+.02;
cf=feplot;cf.model=model;cf.def=def;
fecom(';view1;animtime;ch20');
```

See also `fe_eig`, `fe_time`, `fe_mk`

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 7.1 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.

`feplot` allows the visualization of these energies using a color coding. You can use the high level commands `fecom ColorDataK` or `ColorDataM` or compute energies and initialize color with (see also the `d_ubeam` and `gartup` demos)

```
feplot('ColorDataElt',RESULT.StainE,RESULT.EltId);
fecom(';showpatch;ColorBar')
```

For backward compatibility, `fe_stres` returns `[StrainE,KinE]` as two arguments if no element selection is given. To select all elements, use the `'ener groupall'` command.

Note that the element energy and **not** energy density is computed. This may be misleading when displaying energy levels for structures with uneven meshes. `upcom` provides a compiled version of `fe_stres` for the superelements it handles.

`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).

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`	mean stress at element stress restitution points. Result is a structure with fields `.EltId` and `.data`.

The `fecom ColordataStress` directly calls `fe_stres` and displays the result. For example, run the basic element test `q4p testsurstress`, then display various stresses using

```
q4p testsurstress
fecom('colordatastress atcenter')
fecom('colordatastress mises')
fecom('colordatastress sII atcenter')
```

See also `fe_mk`, `feplot`, `fecom`

fe_super

Purpose Generic element function for superelement support.

Description Superelements are stored in global variables whose name is of the form `SE`*`Name`*. `fe_super` ensures that superelements are correctly interpreted as regular elements during model assembly, visualization, etc. The superelement *`Name`* must differ from all function names in your MATLAB `path`. `d_cms2` demonstrates the use of superelements.

Superelement variables are structure arrays with some or all of the fields described below.

`SE`*`Name`*`.Opt`

Options characterizing the type of superelement as follows.

`Opt(1,1)`	1	**unique** superelements are used only once. In model description matrices they have no associated element property rows.
	2	**generic** superelements are used several times and each occurrence is associated to an element property row.
	3	**FE update** unique superelements (see `upcom`)
`Opt(1,2)`		`ProID` element property identification number of unique superelements
`Opt(1,3)`		estimated maximum matrix word count (for `fe_mk`)
`Opt(1,4)`	1	FE update superelement uses non symmetric matrices
`Opt(2,:)`		matrix types for the superelement matrices. Each non zero value on the second row of `Opt` specifies a matrix stored in the field `SE`*`Name`*`.K{i}` (where `i` is the column number). The value of `Opt(2,i)` indicates the matrix type. Types defined in the *SDT* are `1` stiffness, `2` mass, `3` viscous damping, `4` hysteretic damping, `5` geometric stiffness.
`Opt(3,:)`		is used to define the coefficient associated with each of the matrices declared in row 2. An alternative mechanism is to define an element property in the `il` matrix. If these coefficients are not defined they are assumed to be equal to 1.

Element header rows follow the standard format

```
[Inf abs('Name') 0 EGID EltId]
```

The `EltId` is only used by unique superelements since generic superelements store it in their element rows. **Element rows** for generic superelements follow the format

```
[n1 ... ni MatID ProID EltID]
```

where `n1` to `ni` are node identification numbers (as many as in the `SE`*`Name`*`Node` matrix), `MatID` is unused (generally set to 0) and `ProID` is the identification number of the element properties (matching an identification number in the property declaration matrix `il`).

Material properties are not used by superelements.

Element property rows (in a standard property declaration matrix `il`) for superelements take the form

```
[ProID  coef1 ... coefi]
```

with `ProID` the property identification number and the coefficients allow the creation of a weighted sum of the superelement matrices `SE`*`Name`*`.K{i}`. Thus, if `K{1}` and `K{3}` are two stiffness matrices and no other stiffness matrix is given, the superelement stiffness is given by `coef1*K{1}+coef3*K{3}`.

If `ProID` is not given, `fe_super` will see if `SE`*`Name`*`.Opt(3,:)` is defined and use coefficients stored in this row instead. If this is still not given, all coefficients are set to 1.

`SE`*`Name`*`.Node`

Nominal node matrix. Contains the nodes used by the unique superelement or the nominal generic superelement (see section 7.1).

`SE`*`Name`*`.DOF`

Degree of freedom definition vector. For `unique` superelements (see `SE`*`Name`*`.Opt`), the variable `SE`*`Name`*`.DOF` defines the superelement DOFs. For element DOFs of unique superelements, the element identifier should be `-1` which will be automatically replaced by `-EltId` for assembly (see section 7.5 for details on element DOFs).

For `generic` superelements, `SE`*`Name`*`.DOF` defines a generic DOF definition vector. This vector follows the usual DOF definition 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 the `-1` for element numbers is replaced by the actual number of the element (see `mdof` page 146).

`SE`*`Name`*`.K{i}`

Superelement matrices. The presence and type of these matrices is declared in `SE`*`Name`*`.Opt` (see above). They must all be consistent with the `SE`*`Name`*`.DOF` vector. For generic superelements, they can be expressed in local coordinates if a `SE`*`Name`*`.Ref` coordinate transformation specification is given.

`SE`*`Name`*`.Patch`

Patch face matrix for drawing with `feplot`. The patch face matrix is a matrix of node indices (with respect to the unique or generic node numbers) that describe the patches used to plot a surface deformation of the structure. See section 7.14 and the MATLAB `patch faces` property for more details.

`SE`*`Name`*`.Line`, a vector of node indices (with respect to the unique or generic node numbers) describing the line of nodes that were used to plot wire frame deformations of the structure in earlier versions of the *SDT*, is supported by `fe_super` for backward compatibility with user defined functions.

`SE`*`Name`*`.Elt, SE`*`Name`*`.il, SE`*`Name`*`.node, SE`*`Name`*`.pl`

Initial model retrieval for `unique` superelements. `SE`*`Name`*`.Elt` contains the initial model description matrix which allows the construction of a detailed visualization as well as post-processing operations. `SE`*`Name`*`.Node` contains the nodes used by this model. The `.pl` and `.il` fields store material and element properties for the initial model.

`SE`*`Name`*`.TR`

`SE`*`Name`*`.TR` contains the definition of a possible projection using `mdof`, `adof` and `T` stored in a single matrix using `TR=[Inf adof(:)';mdof T]`.

For a superelement that has not been reduced `mdof` should be empty or identical to `SE`*`Name`*`.DOF`. `adof` and `T` should then be empty.

For a reduced superelement, `adof` and `SE`*`Name`*`.DOF` should match. `T` and `mdof` can then be used to retrieve motion at all DOFs of the unreduced model. Note that this retrieval is not supported for generic superelements. The format for this field is likely to change.

`SE`*`Name`*`.Ref`

Coordinate transformation specification. The matrices of `generic` superelements can be specified in local coordinate systems.

Type 1 transforms are characterized as follows. `SE`*`Name`*`Ref` contains `[1 n1 n2 n3 n4 T0(:)']`. A 3 by 3 local basis matrix T is constructed where the local x axis (first column of T) is collinear to the vector going from node 1 to node 2, the local y axis is collinear to the component orthogonal to x of the vector going from node 3 to node 4, the local z axis is given by the vector product $x \wedge y$. The transformation between T and the initial basis T_0 (the default for T_0 is the global xyz basis) is given by $T_0^T T$.

It is assumed that the DOFs of rotated generic superelements form a sequence of three component vectors (xyz translations or rotations) defined at nodes. Each of these vectors then can be rotated using $q_{global} = \left[T_0^T T\right] q_{local}$. The `fesuper Make`*`Name`*` Complete` ensures that this condition is verified by sorting DOFs and adding DOFs with zero contributions when needed.

The `fesuper SetRef` command can be used to specify a coordinate transformation. A sample application is treated in `d_cms2`.

See also `fesuper`, `upcom`, the `d_cms2` demonstration

fe_time,of_time

Purpose Computation of time response.

Syntax

```
def=fe_time('Command',model,Case,q0)
def=fe_time(com,model,Case,q0,opt)
```

Description fe_time computes the time response given initial conditions, boundary conditions, load case (section 5.2) and time parameters. Linear and non linear problems are supported. The companion mex of_time supports steps that need to be compiled.

Two types of time integration algorithm are possible : the Newmark schemes (newmark command) and the time Discontinuous Galerkin method [45] [46] (dg command). No damping and no non linearities are supported for Discontinuous Galerkin method.

Note that you may access to the fe_time commands graphically with the simulate tab of the feplot GUI. See tutorial (section 5.3) on how to compute a response.

Input arguments are string or data structure commands detailed below, the model and associated Case (containing input force signal).

Initial conditions can also be provided in q0 wit the second column giving velocity if any. If q0 is empty, zero initial conditions are taken.

Accepted solvers are

newmark

Newmark scheme with damping support. If com is string, you may enter simulation informations in the command using the format

```
'Newmark (beta) (gamma) (t0) (deltaT) (Nstep) (Nf)'
```

With beta, gamma the standard Newmark parameters. t0 the initial time, deltaT the fixed time step, Nstep the number of steps, Nf the optional number of time step of the input force.

For example :

```
def=fe_time('newmark .25 .5 0 1e-4 50',model,Case,q0);
```

[beta alpha t0 deltaT Nstep Nf] can also be given as a last input argument opt.

This is a simple 1D example plotting the propagation of the velocity field using a Newmark implicit algorithm :

```
[model,Case]=fe_time('demo bar'); q0=[];
def=fe_time('newmark .25 .5 0 1e-4 100',model,Case,q0);
def_v=def;def_v.def=def_v.v; def_v.DOF=def.DOF+.01;
feplot(model,def_v);
if sp_util('issdt') fecom(';view2;animtime;ch30;scd3');end
```

and here is a 2D example :

```
model=fe_time('demo 2d'); q0=[];
com.Method='newmark';
com.Opt=[.25 .5 3e-4 1e-4 50 10];
com.Residual='';
[def,model]=fe_time(com,model,'Case 1',q0);
if sp_util('issdt')
 cf=feplot;cf.model=model;cf.def=def;
 fecom('colordataa');
 cf.ua.clim=[0 2e-6];fecom(';view2;animtime;ch20;scd1e-2;');
 st=fullfile(getpref('SDT','tempdir'),'test.avi');
 fecom(['animavi ' st])
end

% example to select output DOFs
com.OutInd=fe_c(fe_case(model,'GettDof'), ...
   feutil('findnode y==0',model)+.02,'ind');
[def,model]=fe_time(com,model,'Case 1',q0);

% example to select output time steps
com=rmfield(com,'OutInd');
com.OutputFcn=[11e-4 12e-4];
[def,model]=fe_time(com,model,'Case 1',q0);
```

dg

Discrete Galerkin. Options are [unused unused t0 deltaT Nstep Nf]

This is the same 1D example but using the Discontinuous Galerkin method :

```
[model,Case]=fe_time('demo bar');  q0=[];
def=fe_time('dg .25 .5 0 1e-4 100',model,Case,q0);
def_v=def;def_v.def=def_v.v; def_v.DOF=def.DOF+.01;
feplot(model,def_v);
if sp_util('issdt') fecom(';view2;animtime;ch30;scd3');
else; fecom(';view2;scaledef3'); end
```

NLNewmark

Newmark scheme with damping support. If com is string, you may enter simulation informations in the command using the format

```
'NLnewmark (beta) (gamma) (t0) (deltaT) (Nstep) (Nf)'
```

With beta, gamma the standard Newmark parameters. t0 the initial time, deltaT the fixed time step, Nstep the number of steps.

com

The com data structure has fields

.Method	'newton', 'dg' or 'newmark'
.Opt	[beta alpha t0 deltaT Nstep Nf]
.OutInd	DOF output indices (see 2D example). This selection is based on the state DOFs which can be found using fe_case(model,'GettDof').
.MaxIter	maximum number of iterations
.Jacobian	string to be evaluated to generate a factored jacobian matrix in matrix or ofact object ki. The default string is 'ki=ofact(model.K{3}+2/dt*model.K{2} +4/(dt*dt)*model.K{1});'
.JacobianUpdate	(only for newton) : default is 0, 0 if modified Newton (no update in Newton iterations), 1 if update in Newton iteration
.Residual	The default residual is 'r = model.K{1}*a+model.K{2}*v+model.K{3}*u-fc;'
.InitAcceleration	optional field to be evaluated to initialize the acceleration field.
.OutputFcn	command to be evaluated for post-processing of a time vector containing the output time step
.TimeVector	optional value of time steps
.RelTol	threshold for convergence tests. The default is getpref('OpenFEM','THRESHOLD',1e-6);

of_time

The of_time function is a low level function dealing with CPU and/or memory consuming steps of a time integration.

The commands are

'lininterp'	linear interpolation
'storelaststep'	pre-allocated saving of a time step
'newmarkinterp'	Newmark interpolation (low level call)

The 'lininterp' command which syntax is

`out = of_time ('lininterp',table,val,last)` ,

computes val containing the interpolated values given an input table which first column contains the abscissa and the following the values of each function. Due to performance requirements, the abscissa must be in ascending order. The variable last contains [i1 xi si], the starting index (beginning at 0), the first abscisse and coordinate. The following example shows the example of 2 curves to interpolate:

```
out=of_time('lininterp',[0 0 1;1 1 2;2 2 4],linspace(0,2,10)',[0 0 0])
```

The storelaststep command makes a deep copy of the displacement, celerity and acceleration fields (stored in each column of the variable uva in the preallocated variables u, v and a following the syntax:

`of_time('storelaststep',uva,u,v,a);`

The newmarkinterp command is used by fe_time when the user gives a TimeVector in the command using a Newmark scheme. Given an acceleration vector a1 at time t1 and the uva matrix containing in each column, displacement, celerity and acceleration at the preceding time step t0, it interpolates according to Newmark scheme (see Geradin p.371 eq. 7.3.9) the displacement at time t1.

The low level call of newmarkinterp is

`of_time ('newmarkinterp', out, beta,gamma,uva,a1, t0,t1)`

The out data structure must be preallocated and is a modified input containing the following fields :

`OutInd`	Output indice, must be given
`cur`	`[Step dt]`, must be given
`def`	must be preallocated

See also `fe_mk`, `fe_load`, `fe_case`

fe_var

Purpose Uncertainty propagation tools

Description xxx

Data structures xxx

`desired`

The objectives currently supported are modal frequencies and excitabilities. Excitabilities appear naturally in the definition of transfer functions which, for mass normalized modes, are given by

$$H(\omega, p) = \sum_{j=1}^{N} \frac{[c]\{\phi_j\}\{\phi_j\}^T [b]}{-\omega^2 + 2i\zeta_j\omega_j\omega + \omega_j^2} \tag{9.1}$$

Excitability is thus defined as the contribution of mode j to the transfer function at it's resonance frequency

$$e_j = \frac{[c]\{\phi_j\}\{\phi_j\}^T [b]}{2\zeta_j\omega_j^2\omega} \tag{9.2}$$

To compute excitabilities, one thus needs to compute modal output shape matrices $[c]\{\phi_j\}$. One assumes that a collocated sensor is defined for each input so that the input shape matrix $\{\phi_j\}^T [b]$ can actually be related to an output using the reciprocity assumption (note that supporting non-reciprocal cases would be a major development).

`targ.ftarg`	indices of target frequencies
`targ.outfreq`	frequencies kept for outputs
`targ.outcp`	observation matrix for outputs (modeshape components) to be retained
`targ.trans_pair`	transmissibility pairs a $NT \times 2$ matrix giving the indices of outputs and inputs (actually output recriprocal to the desired input) in `targ.outcp`
`targ.trlab`	a cell array of labels for each transmissibility defined in `targ.trans_pair`

fe_var

This data structure must be defined prior to most uncertainty analyses and is typically stored as an `info` entry in `Up.Stack`.

```
Up=stack_set(Up,'info','fe_var desired',desired);
```

`experiment`

Experiments give a list of possible parameter values. They are characterized by the the fields

`ex.val`	matrix where each rows gives values of all parameters at a particular design point
`ex.param`	indices of the parameters that actually change during the experiment
`ex.edge`	connectivity matrix used to define lines connecting different design points of the experiment

`result`

Results summarize the output of a reanalysis simulation.

`re.freq`	output frequencies with one design point per column
`re.cp`	output shape matrices with one design point per column
`re.trans`	objective transmissibilities
`re.val`	matrix where each rows gives values of all parameters at a particular design point
`re.des`	parameter stack
`re.desired`	data structure defining objectives for the simulation

Commands

`BuildT`

`BuildMC`

`Error`

`GetOutputs`

`Par[Face,Grid,CubeEdge,rand]`

`Par` commands generate standard experiments (series of design points) that can then be used to evaluate model properties at these points. These are defined based on the `upcom` parameter matrix obtained with `upcom('parcoefpar')` where each row describes the acceptable range of a parameter

```
[type cur min max vtype]
```

`fe_var('parface1 2',Up,2:3)` generates a design points at the orthogonal projection of the nominal point (given by `par(:,2)`) on the lower and/or upper faces defined by the parameter range defined by `par(:,3:4)`. `ParFace 1` only generates points of faces with minimum parameter values. Face 0 is the nominal point. The optional third argument (`2:3` in the example) is used to enforce variations on a subset of parameters. The output is an `experiment` data structure described below.

`fe_var('pargrid` *`opt`*`',Up,`*`indp`*`)` generates a uniform grid by dividing the range of each parameter in *`opt`* points. When used for selected parameters by giving *`indp`*, the unused parameters are set to their nominal value.

`fe_var('pargrid` *`opt`* `edge` *`elevel`*`',Up,`*`indp`*`)` generates a uniform grid by dividing the range of each parameter in *`opt`* points. One then only retains points that are on an edge level *`elevel`* defined by the fact that *`elevel`* parameters are equal to their minimal or maximal value.

`fe_var('par cubeedge` *`opt`*`',Up,`*`indp`*`)` the one dimensional edges of the hypercube defined by parameters selected in *`indp`*. xxx

`fe_var('par` *`opt`*`',Up,`*`indp`*`)` creates a random experiment on *`indp`* with *`opt`* design points.

`InitModes`

`fe_var('InitModes` *`ParCommand`*`',Up)` computes full order modes for the experiment defined by *`ParCommand`*. The default is `ParFace 0 2` which keeps the nominal point its projections on the positive faces of the hypercube.

The results are saved in a `Modes.mat` file and are typically used to build reanalysis bases using xxx commands.

`plot[Delaunay,Hist,map&point,Cube]`

Tutorial Uncertainty propagation is supported for parameterized superelements supported by `upcom`. The main steps of an uncertainty analysis are

- Parameterization. This phase is actually supported by `upcom` and you can look at examples under the `upcom par` commands.
- Reanalysis basis creation.

idcom

Purpose UI command functions for standard operations in identification.

Syntax

```
idcom('CommandString');
```

Description `idcom` provides a simple access to standard operations in identification. The way they should be sequenced is detailed in section 3.3 which also illustrates the use of the associated GUI.

`idcom` must be used with the `iiplot` interface for response data visualization as this interface is used to visualize the results of different operations during the identification. `idcom` uses and modifies data found in the standard database wrapper `XF` (see `iiplot`).

The information given below details each command (see the `commode` help for hints on how to build commands and understand the variants discussed in this help). Without arguments `idcom` enters the command mode provided by `commode` and gives direct access to `idcom` and `iicom` commands (`idcom,iicom>` prompt). Information on how to modify standard plots is given under `iicom`.

Commands

`e [ ,`*i* *w*`]`

Single pole narrow-band model identification. `e` calls `ii_poest` to determine a single pole narrow band identification for the data set `IIxf`.

A bandwidth of two percent of *w* is used by default (when *i* is not given). For *i*`<1`, the *i* specifies the half bandwidth as a fraction of the central frequency *w*. For *i* an integer greater than 5, the bandwidth is specified as a number of retained frequency points.

The selected frequency band is centered around the frequency *w*. If *w* is not given, `ii_poest` will wait for you to pick the frequency with your mouse.

If the local fit does not seem very good, you should try different bandwidths (values of *i*).

The results are an estimated pole `IIpo1`, residue matrix `IIres1`, and FRF `IIxe` (which is overlaid to `IIxf` in `iiplot` *drawing axes*). If, based on the plot(s), the estimate seems good it should be added to the current pole set `IIpo` using the `ea` command.

`ea`

Add `IIpo1` *to* `IIpo`. If appropriate poles are present in `IIpo1` (after using the `e` or `f` commands for example) they should be added to the current pole set `IIpo` using the `ea` command. These poles can then be used to identify a multiple pole broadband model (`est` and `eup` commands).

If all poles in `IIpo1` are already in `IIpo`, the two are only combined when using the `eaf` command (this special format is used to prevent accidental duplication of the nodes).

`er [num` *`i`*`, f` *`w`*`]`

Remove poles from `IIpo`. The poles to be removed can be indicated by number using `'er num` *`i`*`'` or by frequency using `'er f` *`w`*`'` (the pole with imaginary part closest to *w* is removed). The removed pole is placed in `IIpo1` so that an `ea` command will undo the removal.

`est`

Broadband multiple pole identification without pole update. `est` uses `id_rc` to identify a model based on the complete frequency range. This estimate uses the current pole set `IIpo` but does not update it. The results are a residue matrix `IIres`, and corresponding FRF `IIxe` (which is overlaid to `IIxf` in `iiplot` *drawing axes*). In most cases the estimate can be improved by optimizing the poles using the `eup` or `eopt` commands.

`eup` *`dstep fstep`* `[local, num` *`i`* `]`

Update of poles. `eup` uses `id_rc` to update the poles of a multiple pole model based data within `IDopt.SelectedRange`. This update is done through a non-linear optimization of the pole locations detailed in section 3.3.2. The results are updated poles `IIpo` (the initial ones are stored in `IIpo1`), a residue matrix `IIres`, and corresponding FRF `IIxe` (which is overlaid to `IIxf` in `iiplot` *drawing axes*).

In most cases, `eup` provides significant improvements over the initial pole estimates provided by the `e` command. In fact the only cases where you should not use `eup` is when you have a clearly incomplete set of poles or have reasons to suspect that the model form used by `id_rc` will not provide an accurate broadband model of your response.

Default values for damping and frequency steps are `0.05` and `0.002`. You may specify other values. For example the command `'eup 0.05 0.0'` will only update damping values.

It is often faster to start by optimizing over small frequency bands while keeping all the poles. Since some poles are not within the selected frequency range they should not be optimized. The argument `local` placed after values of *dstep* and *fstep* (if any) leads to an update of poles whose imaginary part are within the retained frequency band.

When using local update, you may get warning messages about conditioning. These just tell you that residues of modes outside the band are poorly estimated, so that the message can be ignored. While algorithms that by-pass the numerical conditioning warning exist, they are slower and don't change results so that the warning was left.

In some cases you may want to update specific poles. The argument `num` *i* where *i* gives the indices in `IIpo` of the poles you want to update. For example `'eup 0.0 0.02 num 12'` will update the frequency of pole `12` with a step of 2%.

- The poles in `IIpo` are all the information needed to obtain the full model estimate. You should save this information in a text or `.mat` file regularly to be able to restart/refine your identification.
- You can get a feel for the need to further update your poles by showing the error and quality plots (see `iiplot` and section 3.3.1).

`eopt [local, num` *i* `]`

Update of poles. `eopt` is similar to `eup` but uses `id_rcopt` to optimize poles. `eopt` is often more efficient when updating one or two poles (in particular with the `eopt local` command after selecting a narrow frequency band). `eopt` is guaranteed to improve the quadratic cost (4.4) so that using it rarely hurts.

`find`

Find a pole. This command detects minima of the MMIF that are away from poles of the current model (`IIpo`) and calls `ii_poest` to obtain a narrow band single pole estimate in the surrounding area. This command can be used as an alternative to indicating pole frequencies with the mouse (`idcom e` command). More complex automated model initialization will be introduced in the future.

`f` *`i`*

Graphical input of frequencies. `f` *`i`* prompts the user for mouse input of *`i`* frequencies (the abscissa associated with each click is taken to be a frequency). The result is stored in the pole matrix `IIpo1` assuming that the indicated frequencies correspond to poles with 1% damping. This command can be used to create initial pole estimates but the command `e` should be used in general.

`dspi` *`nm`*

Direct system parameter identification. `dspi` uses `id_dspi` to create a *`nm`* pole state space model of `IIxf`. `nm` must be less than the number of sensors. The results are transformed to the residue form which gives poles `IIpo1`, a residue matrix `IIres1`, and corresponding FRF `IIxe` (which is overlaid to `IIxf` in `iiplot` *drawing axes*).

`mass` *`i`*

Computes the generalized mass at address *`i`*. If the identified model contains complex residues (`IDopt.Fit='Pos'` or `'Complex'`), `res2nor` is used to find a real residue approximation. For real residues, the mass normalization of the mode is given by the fact that for collocated residues reciprocity implies

$$c_{Col}\phi_j = \phi_j^T b_{Col} = \sqrt{R_{jCol}} = (m_{jCol})^{-1/2}$$

The mass at a given sensor *`i`* is then related to the modal output $c_l\phi_j$ of the mass normalized mode by $m_{lj} = (c_l\phi_j)^{-2}$. This command can only be used when collocated transfer functions are specified and the system is assumed to be reciprocal (see `idopt`).

`poly` *`nn nd`*

Orthogonal polynomial identification. `poly` uses `id_poly` to create a polynomial model of `IIxf` with numerators of degree *`nn`* and denominators of degree *`nd`*. The corresponding FRFs are stored in `IIxe` (which is overlaid to `IIxf` in `iiplot` *drawing axes*).

`[Table,Tex] IIpo`

Formatted printout of pole variables `IIpo` or `IIpo1`. With the `Tex` command the printout is suitable for inclusion in LATEX.

This command is also accessible from the idcom figure context menu.

See also idcom, iicom, iiplot, id_rc, section 3.3

idopt

Purpose handling of options used by the identification related routines.

Syntax

```
idopt
iop2 = idopt
IDopt.OptName = OptValue;
```

Description `idopt` with no argument sets default options based `XF(1)` (see `iiplot` and `xfopt`). These options are stored in the standard global variable `IDopt` and can be edited using the `Options` tab in the `idcom` GUI figure.

`iop2 = idopt` returns a *SDT* `handle` to a set options that may differ from those of the global variable `IDopt`.

`IDopt=idopt` checks the standard global variable `IDopt`. `idopt('default')` reinitializes `IDopt` based on the first data set in the standard database wrapper `XF`.

The display of an identification option variable (type `IDopt` at the MATLAB prompt for example) gives a detailed list of the options

```
IDopt (global variable) =
   ResidualTerms : [ 0 | 1 (1) | 2 (s^-2) | {3 (1 s^-2)} | 10 (1 s)]
   DataType : [ {disp./force} | vel./force | acc./force ]
   AbscissaUnits : [ {Hz} | rd/s | s ]
   PoleUnits : [ {Hz} | rd/s ]
   SelectedRange : [ 1-3124 (4.0039-64.9998) ]
   FittingModel : [ Posit. cpx | {Complex modes} | Normal Modes]
   NSNA : [ 0 sensor(s) 0 actuator(s) ]
   Reciprocity : [ {Not used} | 1 FRF | MIMO ]
   Collocated : [ none declared ]
```

with the currently selected value shown between braces `{ }`.

SDT `handle` overloads the MATLAB `getfield` and `setfield` commands so that you can easily access each option. `IDopt.OptName` displays the associated option value using the format shown above. `IDopt.OptName=OptValue` sets the option. `OptName` need only specify enough characters to allow a unique option match. Thus `IDopt.res` and `IDopt.ResidualTerms` are equivalent. Typical option sets would be

```
 IDopt.res = 2; IDopt.sel=[1 1024]; IDopt.Po='Hz';
```

idopt

The following is a list of possible options with indications as to where they are stored. Thus `IDopt.res=2` is simply a user friendly form for the old call `IDopt(6)=2` which you can still use.

`Res`		Residual terms (stored in `IDopt(1)`)
	`0`	none
	`1`	Static correction (high frequency mode correction)
	`2`	Roll-off (s^{-2}, low frequency mode correction).
	`3`	Static correction and roll-off (default)
	`10`	*1* and s, this correction is only supported by `id_rc` and should be used for identification in narrow bandwidth (see `ii_poest` for example)
	`-`i	An alternate format uses negative numbers with decades indicating powers (starting at s^{-2}). Thus `Ass=-1101` means an asymptotic correction with terms in $s^{-2}, 1, s$
`Data`		type (stored in `IDopt(2)`)
	`0`	displacement/force (default)
	`1`	velocity/force
	`2`	acceleration/force
`Abscissa`		units for vector `w` can be Hz, rad/s or seconds
`Pole`		units can be Hz or rad/s
		units are actually stored in `IDopt(3)` with units giving abscissa units (`01 w` in Hertz, `02 w` in rad/s, `03 w` time seconds) and tens pole units (`10 po` in Hertz, `20 po` in rad/s). Thus `IDopt (3) = 12` gives `w` in rad/sec and `po` in Hz.
`Selected`		frequency range indices of first and last frequencies to be used for identification or display (stored in `IDopt(4:5)`)
`Fitting`		model (see `res` page 37, stored in `IDopt(6)`)
	`0`	positive-imaginary poles only, complex mode residue
	`1`	complex mode residue, pairs of complex-conjugate poles (default)
	`2`	normal mode residue
`ns,na`		number of sensors/actuators (outputs/inputs) stored in `IDopt(7:8)`)

Recip		method selection for the treatment of reciprocity (stored in IDopt(12))
	1	means that only iC1 (IDopt(13)) is declared as being collocated. id_rm assumes that only this transfer is reciprocal even if the system has more collocated FRFs
	na	(number of actuators) is used to create fully reciprocal (and minimal of course) MIMO models using id_rm. na must match non-zero values declared in iCi.
	-nc	(with nc the number of collocated FRFs) is used to declare collocated FRFs while not enforcing reciprocity when using id_rm.
iC1 ...		indices of collocated transfer functions in the data matrix (see the xf format page 40)

See also xfopt, idcom, iiplot

id_dspi

Purpose Direct structural system parameter identification.

Syntax

```
[a,b,c,d] = id_dspi(y,u,w,IDopt,np)
```

Description The direct structural system parameter identification algorithm [47] considered here, uses the displacement frequency responses $y(s)$ at the different sensors corresponding to the frequency domain input forces $u(s)$ (both given in the `xf` format). For example in a SIMO system with a white noise input, the input is a column of ones `u=ones(size(w))` and the output is equal to the transfer functions `y=xf`. The results of this identification algorithm are given as a state-space model of the form

$$\left\{ \begin{array}{c} \dot{p} \\ \ddot{p} \end{array} \right\} = \left[\begin{array}{cc} 0 & I \\ -K_T & -C_T \end{array} \right] \left\{ \begin{array}{c} p \\ \dot{p} \end{array} \right\} + \left[\begin{array}{c} 0 \\ b_T \end{array} \right] \{u\} \quad \text{and} \quad \{y\} = \left[\begin{array}{cc} c_T & 0 \end{array} \right] \left\{ \begin{array}{c} p \\ \dot{p} \end{array} \right\}$$

where the pseudo-stiffness K_T and damping C_T matrices are of dimensions `np` by `np` (number of normal modes). The algorithm, only works for cases where `np` is smaller than the number of sensors (`IDopt.ns`).

For SIMO tests, normal mode shapes can then be obtained using `[mode,freq] = eig(-a(np+[1:np],1:np))` where it must be noted that the modes **are not** mass normalized as assumed in the rest of the *Toolbox* and thus cannot be used directly for predictions (with `nor2xf` for example). Proper solutions to this and other difficulties linked to the use of this algorithm (which is provided here mostly for reference) are not addressed, as the main methodology of this *Toolbox* (`id_rc`, `id_rm`, and `id_nor`) was found to be more accurate.

For MIMO tests, `id_dspi` calls `id_rm` to build a MIMO model.

The identification is performed using data within `IDopt.SelectedRange`. `y` is supposed to be a displacement. If `IDopt.DataType` gives `y` as a velocity or acceleration, the response is integrated to displacement as a first step.

See also `idopt`, `id_rc`, `id_rm`, `psi2nor`, `res2nor`

id_nor

Purpose Identification of normal mode model, with optimization of the complex mode output shape matrix.

Syntax

```
NOR                 = id_nor(XF(5))
NOR                 = id_nor( ... )
[om,ga,phib,cphi]   = id_nor( ... )
[new_res,new_po]    = id_nor( ... )
[ ... ]             = id_nor(res,po,IDopt)
[ ... ]             = id_nor(res,po,IDopt,ind,opt,res_now)
```

Description `id_nor` is meant to provide an optimal transformation (see details in [5] or section 3.4.3) between the residue (result of `id_rc`) and non-proportionally damped normal mode forms

$$\{y(s)\} = \sum_{j=1}^{2N} \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j}\{u\} \quad \text{and} \quad \begin{array}{c}\left[Is^2+\Gamma s+\Omega^2\right]\{p\} = \left[\phi^T b\right]\{u\} \\ \{y\} = [c\phi]\{p\}\end{array}$$

The output arguments are either

- the standard normal mode model `freq,ga,phib,cphi` (see `nor` page 28) when returning 4 outputs
- the associated normal model data structure `NOR` when returning one output
- or the residues of the associated model `new_res` and poles `po` (see `res` page 37) when returning 2 outputs. With this output format, the residual terms of the initial model are retained.

The algorithm combines `id_rm` (which extracts complex mode output shape matrices $c\psi$ from the residues `res` and scales them assuming the system reciprocal) and `psi2nor` (which provides an optimal second order approximation to the set of poles `po` and output shape matrices $c\psi$).

Since the results of `psi2nor` can quite sensitive to small errors in the scaling of the complex mode outputs $c\psi$, an optimization of all or part (using the optional argument `ind` to indicate the residues of which poles are to be updated) collocated residues can be performed. The relative norm between the identified residues `res` and those of the normal mode model is used as a criterion for this optimization.

Three optimization algorithms can be selected using `opt` (1: `id_min` of the *Structural Dynamics Toolbox*, 2: `fmins` of MATLAB, 3: `fminu` of the *Optimization Toolbox*). You can also restart the optimization using the residues `old_res` while still comparing the result with the nominal `res` using the call

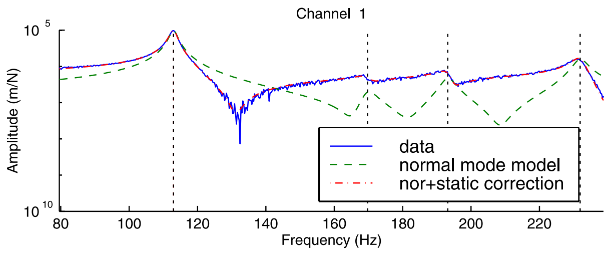
```
[new_res,po] = id_nor(res,po,IDopt,ind,opt,old_res)
```

Notes

id_nor is only defined if IDopt.Reciprocity is 1 FRF or MIMO (12) and for cases with more sensors than modes (check IDopt.NSNA). id_nor may not work for identifications that are not accurate enough to allow a proper determination of normal mode properties.

In cases where id_nor is not applicable, normal mode residues can be identified directly using id_rc with IDoptFit='Normal' or an approximate transformation based on the assumption of proportional damping can be obtained with res2nor.

id_nor does not handle cases with more poles than sensors. In such cases res2nor can be used for simple approximations, or id_nor can be used for groups of modes that are close in frequency.



Residual terms can be essential in rebuilding FRFs (see figure above taken from demo_id) but are not included in the normal mode model (freq, ga, phib, cphi). To include these terms you can use either the residues new_res found by id_nor

```
xf = res2xf(new_res,po,w,IDopt)
```

or combine calls to nor2xf and res2xf

```
xf = nor2xf(om,ga,phib,cphi,w) + ...
     res2xf(res,po,w,IDopt,size(po,1)+1:size(res,1))
```

Example

```
gartid
if XFdof(4,2)~=1012.03;% Needed to have positive driving point FRFs
 IIxf=-IIxf; XFdof(:,2)=1012.03; idcom('est');
end
nor = id_nor(XF(5));
XF(3)=nor2xf(nor,IIw,'hz struct acc');
iicom('iixhon')
```

See also

id_rc, res2nor, id_rm, psi2nor, demo_id

id_poly

Purpose Parametric identification using `xf`-orthogonal polynomials.

Syntax

```
[num,den] = id_poly(xf,w,nn,nd)
[num,den] = id_poly(xf,w,nn,nd,IDopt)
```

Description A fit of the provided frequency response function `xf` at the frequency points `w` is done using a rational fraction of the form $H(s) = num(s)/den(s)$ where `num` is a polynomial of order `nn` and `den` a polynomial of order `nd`. The numerically well conditioned algorithm proposed in Ref. [16] is used for this fit.

If more than one frequency response function is provided in `xf`, the numerator and denominator polynomials are stacked as rows of `num` and `den`. The frequency responses corresponding to the identified model can be easily evaluated using the command `qbode(num,den,w)`.

The identification is performed using data within `IDopt.SelectedRange`. The `idcom poly` command gives easy access to this function.

See also `id_rc`, `invfreqs` of the *Signal Processing Toolbox*.

id_rc, id_rcopt

Purpose

Broadband pole/residue model identification with the possibility to update an initial set of poles.

Syntax

```
[res,po,xe]     = id_rc   (xf,po,w,IDopt)
[res,new_po,xe] = id_rc   (xf,po,w,IDopt,dst,fst)
[res,new_po,xe] = id_rcopt(xf,po,w,IDopt,step,indpo)
```

Description

This function is typically accessed using the idcom GUI figure as illustrated in section 3.3.

For a given set of poles po (see ii_pof for the format), id_rc(xf,po,w,IDopt) identifies the residues of a broadband model, with poles po, that matches the FRFs xf at the frequency points w. (This is implemented as the idcom est command).

As detailed in section 3.3, the poles can (and should) be tuned [13] using either id_rc (ad-hoc dichotomy algorithm, accessible through the idcom eup command) or id_rcopt (gradient or conjugate gradient minimization, accessible through the idcom eopt command). id_rc performs the optimization when initial step sizes are given (see details below).

After the identification of a model in the residue form with id_rc, other model forms can be obtained using id_rm (minimal/reciprocal residue model), res2ss (state-space), res2xf (FRF) and res2tf (polynomial), id_nor (normal mode model).

The different input and output arguments of id_rc and id_rcopt are

xf

Measured data stored in the xf format where each row corresponds to a frequency point and each column to a channel (actuator/sensor pair).

Although it may work for other types of data, id_rc was developed to identify model properties based on *transfer functions from force actuators to displacement sensors.* IDopt(2) lets you specify that the data corresponds to velocity or acceleration (over force always). An integration (division by $s = j\omega$) is then performed to obtain displacement data and a derivation is performed to output estimated FRFs coherent with the input data (the residue model always corresponds to force to displacement transfer functions).

The phase of your data should loose 180^o phase after an isolated lightly damped but stable pole. If phase is gained after the pole, you probably have the complex conjugate of the expected data.

If the experimental set-up includes time-delays, these are not considered to be part of the mechanical system. They should be removed from the data set `xf` and added to the final model as sensor dynamics or actuator dynamics . You can also try to fit a model with a real poles for Pade approximations of the delays but the relation between residues and mechanical modeshapes will no longer be direct.

`w`

Measurement frequencies are stored as a column vector which indicates the frequencies of the different rows of `xf`. `IDopt(3)` is used to specify the frequency unit. By default it is set to `11` (FRF and pole frequencies in Hz) which differs from the *SDT* default of *rad/s* used in functions with no frequency unit option. It is assumed that frequencies are sorted (you can use the MATLAB function `sort` to order your frequencies).

`po, new_po`

Initial and updated pole sets. `id_rc` estimates residues based on a set of poles `po` which can be updated (leading to `new_po`). Different approaches can be used to find an initial pole set

- create narrow-band single pole models (`ii_poest` available as the `idcom e` command)
- pick the pole frequencies on plots of the FRF or MMIF and use arbitrary but realistic values (e.g. 1%) for damping ratios (`ii_fin` available as the `idcom f` command)
- use pole sets generated by any other identification algorithm (`id_poly` and `id_dspi` for example)

Poles can be stored using different formats (see `ii_pof`) and can include both conjugate pairs of complex poles and real poles. (`id_rc` uses the frequency/damping ratio format).

The `id_rc` algorithms are meant for iterations between narrow-band estimates, used to find initial estimates of poles, and broadband model tuning using `id_rc` or `id_rcopt`. These iterations are easier to perform if you save your current pole locations in a text file. For example you should have something like

```
IIpo = [
   1.1298e+02   1.0009e-02
   1.6974e+02   1.2615e-02
   1.9320e+02   1.0457e-02
   2.3190e+02   8.9411e-03];
```

saved in a text editor so that you can paste this set of poles into your MATLAB session. If these are your best poles, `id_rc` will directly provide the optimal residue model. If you are still iterating you may replace these poles by the updated ones or add a pole that you might have omitted initially.

`IDopt`

Identification options (see `idopt` for details). Options used by `id_rc` are `Residual`, `DataType`, `AbscissaUnits`, `PoleUnits`, `SelectedRange` and `FittingModel`.

The definition of channels in terms of actuator/sensor pairs is only considered by `id_rm` which should be used as a post-treatment of models identified with `id_rc`.

`dstep, fstep` (for `id_rc`)

Damping and frequency steps. To update pole locations, the user must specify initial step sizes on the frequency and damping ratio (as fractions of the initial values). `id_rc` then uses the gradient of the quadratic FRF cost to determine in which direction to step and divides the step size by two every time the sign changes. This approach allows the simultaneous update of all poles and has proved over the years to be extremely efficient.

For lightly damped structures, typical step values (used by the `idcom` command `eup`) are 10% on all damping ratios (`dstep = 0.1`) and 0.2% on all frequencies (`fstep = 0.002`). If you only want to update a few poles `fstep` and `dstep` can be given as vectors of length the number of poles in `po` and different step values for each pole.

`idcom('eup 0.05 0.002 local')` can be used to specify `dstep` and `fstep`. The optional `local` at the end of the command specifies that zero steps should be used for poles whose resonance is outside the selected frequency band.

`step, indpo` (for `id_rcopt`)

Methods and selected poles. `step` specifies the method used for step length, direction determination method, line search method, reference cost and pole variations. You should use the default values (empty `step` matrix). `indpo` gives the indices of poles to be updated (`po(indpo,:)` for poles in format 2 are the poles to be updated, by default all poles are updated).

The `idcom eopt` command can be used to access `id_rcopt`. `eoptlocal` calls `id_rcopt` with `indpo` set to only update poles whose resonance is within the selected frequency band.

`res`

Residues are stored in the `res` format (see section 2.6). If the options `IDopt` are properly specified this model corresponds to force to displacement transfer functions (even if the data is acceleration or velocity over force). Experts may want to mislead `id_rc` on the type of data used but this may limit the achievable accuracy.

`xe`

Estimated FRFs correspond to the identified model with appropriate derivation if data is acceleration or velocity over force.

See also `idcom`, `id_rm`, `res2xf`, `res2ss`
Tutorial section section 3.3
`gartid` and `demo_id` demonstrations

id_rm

Purpose Create minimal models of MIMO systems and apply reciprocity constraints to obtain scaled modal inputs and outputs.

Syntax

```
OUT = id_rm(IN,multi)
[psib,cpsi,new_res,new_po] = id_rm(res ,po,IDopt)
[phib,cphi,new_res,new_po] = id_rm(Rres,po,IDopt)
[psib,cpsi,new_res,new_po] = id_rm(res ,po,IDopt,multi)
```

Description `id_rm` is more easily called using the `idcom` GUI figure `Postprocessing` tab, see section 3.4.

`IN` is a data structure (see `xfopt` shapes at DOFs). Required fields are `IN.res` residues, `IN.po` poles, and `IN.idopt` identification options. Options used by `id_rm` are `.FittingModel` (Posit, Complex or Normal modes), `.NSNA` (number of sensors/actuators), `.Reciprocity (not used, 1 FRF or true MIMO)`, `.Collocated` (indices of colloc. FRF when using reciprocity).

`multi` is an optional vector giving the multiplicity for each pole in `IN.po`.

`OUT` is a structure with fields (this format is likely to change in the future)

`.po`	poles with appropriate multiplicity
`.def`	output shape matrix (CPSI)
`.DOF`	Sensor DOFs at which .DEF is defined
`.psib`	input shape matrix (PSIB)
`.CDOF`	indices of collocated FRFs
`.header`	header (5 text lines with a maximum of 72 characters)

The low level calls giving `res`, `po` and `IDopt` as arguments are obsolete and only maintained for backward compatibility reasons.

As shown in more detail in section 3.4, the residue matrix R_j of a single mode is the product of the modal output by the modal input. For a model in the residue form (residue `res`, poles `po` and options `IDopt` identified using `id_rc` for example), `id_rm` determines the modal input `psib` and output `cpsi` matrices such that

$$[\alpha(s)] = \sum_{j=1}^{2N} \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j} \approx \sum_{j=1}^{2N} \frac{[R_j]}{s-\lambda_j} \qquad (9.3)$$

The residues can be either complex mode residues or normal mode residues. In that case the normal mode input `phib` and output `cphi` matrices are real.

The `new_res` matrix is the minimal approximation of `res` corresponding to the computed input and output matrices. `id_rm` uses the number of sensors `IDopt(7)` and actuators `IDopt(8)`.

For MIMO systems (with the both the number of sensors `IDopt(7)` and actuators `IDopt(8)` larger than 1), a single mode has only a single modal output and input which implies that the residue matrix should be of rank 1 (see section 3.4.1). Residue matrices identified with `id_rc` do not verify this rank constraint. A minimal realization is found by singular value decomposition of the identified residue matrices. The deviation from the initial model (introduced by the use of a minimal model with isolated poles) is measured by the ratio of the singular value of the first deleted dyad to the singular value of the dyad kept. For example the following output of `id_rm`

```
 Po #   freq    mul    Ratio of singular values to maximum
   1   7.10e+02  2   :  0.3000 k  0.0029
```

indicates that the ratio of the second singular value to the first is significant (0.3) and is kept, while the second dyad can be neglected (0.0029).

For a good identification, the ratios should be small (typically below 0.1). Large ratios usually indicate poor identification and you should update the poles using `id_rc` in a broad or narrow band update. Occasionally the poles may be sufficiently close to be considered as multiple and you should keep as many dyads as the modal multiplicity using the input argument `multi` which gives the multiplicity for each pole (thus the output shown above corresponds to a multiplicity of `2)`.

`id_rm` also enforces **reciprocity** conditions in two cases

- `IDopt(12)=1`. One transfer function is declared as being collocated. Reciprocity is only applied on the input and output coefficients linked to the corresponding input/output pair.
- `IDopt(12)=na`. As many collocated transfer functions as actuators are declared. The model found by `id_rm` is fully reciprocal (and minimal of course).
- in other cases `IDopt(12)` should be either `0` (no collocated transfer) or equal to `-nc` (`nc` collocated transfers but reciprocal scaling is not desired).

It is reminded that for a reciprocal system, input and output shape matrices linked to collocated inputs/outputs are the transpose of each other ($b = c^T$). Reciprocal scaling is a requirement for the determination of non-proportionally damped normal mode models using id_nor.

In MIMO cases with reciprocal scaling, the quality indication given by id_rm is

```
Po#     freq    mul     sym.    rel.e.
  1    7.10e+02  2 :   0.0038   0.0057
```

which shows that the identified residue was almost symmetric (relative norm of the anti-symmetric part is 0.0038), and that the final relative error on the residue corresponding to the minimal and reciprocal MIMO model is also quite small (0.0057).

Warnings

- id_rm is used by the functions: id_nor, res2nor, res2ss
- Collocated force to displacement transfer functions have phase between 0 and -180 degrees, if this is not true you cannot expect the reciprocal scaling of id_rm to be appropriate and should not use id_nor.
- id_rm only handles complete MIMO systems with NS sensors and NA actuators.

See also idcom, id_rc, id_nor, the demo_id demonstration

iicom

Purpose *UI command* function for FRF data visualization.

Syntax

```
iicom
iicom CommandString
message = iicom('CommandString','TailChar')
```

Description `iicom` is a standard *UI command* function which performs operations linked to the FRF data visualization within the `iiplot` interface. Commands (see the `commode` help for hints on how to build commands and understand the variants discussed in this help) are text strings telling `iicom` what to do. When needed, `iicom` takes data from the standard database wrapper `XF` which points to the global variables linked to FRF data visualization (`IIw`, `IIxf`, `IIxe`, `IIxh`, `IIxi`, see `xfopt` for details).

`iicom` does not modify data except for the frequency band indices (`IDopt.Select` stored in `IDopt(4:5)`) which are modified by the `w` commands. A list of commands available through `iicom` is given below. These commands provide significant extensions to capabilities given by the menus and buttons of the `iiplot` *command figure.*

Commands

`;`

Command chaining. Commands with no input (other than the command) or output argument, can be chained using a call of the form `iicom(';Com1;Com2')`. `commode` is then used for command parsing.

Command function input. This group of buttons is used to give direct access to command functions (without using on-line calls to the UI command functions or the command mode of `commode`). The button indicates which command function is active. If showing `id`, commands are sent to `idcom` or `iicom` (as appropriate). If showing `fe`, commands are sent to `fecom` or `iicom`. If showing `me`, commands are sent to `femesh` or `fecom`. Text commands detailed below can be typed in the text area and are executed at a carriage return (on some platforms when selecting another figure too).

`ad` *i*, `adc` *i*

Display addresses i. Only a restricted number of *channels* (columns of `IIxf`, `IIxe`, `IIxh`, `IIxi`) are displayed in the *drawing axes*. *Addresses* are arbitrary integer numbers used to identify channels. Addresses are stored in the third column of the `XFdof` matrix which has as many rows as there are channels (see `xfopt`). `ad` / `adc` respectively define the addresses (and find the corresponding channels) to be displayed in all / the current drawing axes. The vector of addresses is defined by evaluating the string *i*. For example `ad 101`, displays the channel with *address* 101 in all `axes`.

`cax` *i*, `ca+`,

Change current `axes`. `cax` *i* makes the axis *i* (an integer number) current. `ca+` makes the next axis current. For example, `iicom(';cax1;show rea;ca+;show ima')` displays the real part of the current FRFs in the first axis and their imaginary part in the second. (See also the `sub` command). The button indicates the number of the current axis. Pressing the button executes the `ca+` command.

`ch+`, `ch-`, `ch[+,-]`*i*,

Next/Previous. These commands/buttons are used to scan through plots of the same kind. For `iiplot` axes, this is applied to the current FRFs (columns of `IIxf`, `IIxe`, `IIxh`, `IIxi`) or pole/deformation (rows of `IIres/IIpo` or `IIres1/IIpo1`). For `feplot` axes, the current deformation is changed. You can also increment/decrement channels using the `+` and `-` keys when the current axis is a plot axis or increment by more than 1 using `iicom('ch+`*i*`')`.

`ch` *i*, `chc` *i*

Display channels/poles/deformations i. *Channels* refer to columns of `IIxf`, `IIxe`, `IIxh`, `IIxi`), poles (rows of `IIres`, `IIres1`) or deformations which are displayed in the *drawing axes*. `ch` / `chc` respectively define the indices of the channels to be displayed in all / the current drawing axes. The vector of indices is defined by evaluating the string *i*. For example `iicom ch[1:3]`, displays channels 1 to 3 in all `axes`.

`ga` *i*

Get handle to a particular axis. This is used to easily modify handle graphics properties of `iiplot` axes accessed by their number. For example, you could use `set(iicom('ga1:2'),'xgrid','on')` to modify the grid property of `iiplot` axes 1 and 2.

If you use more than one figure `feplot` or `iiplot` figure, you will prefer the calling format `cf=iiplot; set(cf.ga(1:2),'xgrid','on')`.

`head [Main,Text,Clear]`

Figure header utilities. Header axes are common to all plot functions and span the full figure area (normalized position `[0 0 1 1]`). The *SDT* also tries to keep this on top to that the information is always visible. `iimouse` also raises it when you click outside the area covered by other axes. Any information that you would want to be present on all your plots should be plotted on this axis.

`iicom('HeadMain')` defines the main header title based on the first database header `XF(1).header` (see `xfopt`). You can also specify a string and additional properties that are valid for MATLAB text objects. For example,

```
iicom('head main','Main Title','fontsize',20,'fontname','Times')
```

`iimouse` supports modifications (font, string, position) of this title by providing a context menu (click on the text using the right mouse button). For command based modifications of this object, you can also get its handle using `findobj(cf.head, 'tag', 'iimain')` (where `cf` is a *SDT* `handle` to the `iiplot` or `feplot` figure).

You can put additional text strings in the axis using `iicom('HeadText')` with the same format as `HeadMain`. These objects are created with the `iitext` tag.

`iicom('HeadClear')` deletes all objects from the header axis of the current figure. To find the pointer to the header axis of a given figure use `cf.head`. This can be useful if you want to add other objects that just text.

Note: the obsolete `HeadAdLabel` command is by the use of user defined titles (see the `iicom TitOpt` command).

`IIpo [ ,1], cIIpo[ ,1,tog]`

Poles/residues displayed. `iiplot` displays either the main identified model (global variables `IIpo`, `IIres`) or the alternate model (global variables `IIpo1`, `IIres1`). These commands tell `iiplot` which set to show. By adding a `c` in front of the command (`cIIpo` for example), the choice is only applied to the current axis. You can also toggle which of `IIpo` or `IIpo1` is shown using the `Variables:IIpo` menu (`ciitog` command).

IIx[f,e,h,i] [On,Off]

Global data set displayed. IIxfOn indicates that IIxf should be visible in all axes. IIxfOff turns the visibility of IIxf off. IIxf toggles the visibility of IIxf. The interface handles four sets of data simultaneously IIxf (assumed to be the measurement data), IIxe (identified model FRFs in idcom), IIxh, and IIxi. By adding a c in front of the command (cIIxf for example), the choice is only applied to the current axis. You can also toggle which of the data sets are shown using the Variables:IIx*i* menu.

info

Information on the current state of the interface. This is equivalent to cf=iiplot; disp(cf).

PoleLine [,c] [,3]

Pole line display. By itself, PoleLine toggles the state of pole line display (vertical lines showing the frequencies of poles in IIpo in white and IIpo1 in red). The c applies the command to the current axis only. PoleLine3 places the lines on the pole norm rather than imaginary part used by default (this corresponds to the ii_plp formats 2 and 3).

The state of the current axis (if it is an iiplot axis) can also be changed using the IIplot:PoleLine menu (PoleLineTog command).

Print, Preprint

Print prints the figure of the current iiplot or feplot axis. It changes the current figure if necessary and adds -noui to the print options before calling the MATLAB print command.

Preprint sets the size of the current figure to the physical PaperSize. This is useful to make sure that what you see on the screen is what you will print. In particular legends are based on physical sizes so that the output you get is dependent on the window size.

Show Type,

Specify the current axis type. The iiplot plot functions support a number of plot types (see below, iiplot, and feplot for details) which can be selected using the Show or the PlotType button menu of the iiplot *command figure*.

`Show` gives an easier access to the different plot types through their name as summarized below.

`iiplot` supports the following plots

`abs`	amplitude (also `mag`)	`phu`	unwrapped phase
`pha`	wrapped phase	`mmi`	MMIF
`rea`	real part	`fmi`	forces of MMIF
`ima`	imaginary part	`lny`	local Nyquist
`r&i`	real and imaginary	`ami`	alternate mode indicator
`nyq`	Nyquist plot	`sum`	sum of all FRFs
`cmi`	Complex Mode Indicator	`sumi`	imaginary part sum
`pol`	poles	`rre`	real residue
`fre`	freq. vs. damping	`err`	Nyquist error
`cre`	complex residue	`qual`	Quality plot

`sub [MagPha, i j k[ ,nd][ ,step]]`

This command is the entry point to generate multiple drawing axes within the same figure.

`iicom sub` by itself checks all current axes and fixes anything that is not correctly defined.

`SubMagPha` gives a standard subdivision showing a large amplitude plot and a small wrapped phase plot below. `SubIso` gives a standard 2 by 2 subdivision showing four standard 2-D projections of a 3-D structure.

`Sub i j k` divides the figure in the same manner as the MATLAB `subplot` command. If *k* is set to zero all the *i* times *j* axes of the subplot division are created. Thus the default call to the `sub` command is `'sub 2 1'` which creates two axes in the current figure. If *k* is non zero only one of these axes is created as when calling `subplot(i,j,k)`.

As the `subplot` function, the `sub` command deletes any axis overlapping with the new axis. You can prevent this by adding `nd` to the command string.

The optional `step` modifier increments the deformation shown in each subplot. This is generally used to show various modeshapes in the same figure.

Standard subdivisions are accessible by the `IIplot:sub commands` menu.

Note that `set(cf.ga(i),'position',Rect)` will modify the position of `iiplot` axis `i`. This axis will remain in the new position for subsequent refreshing with `iiplot`.

`TitOpt [ ,c]`*i*

Automated title/label generation options. `Titopt`*i* sets title options for all axes to the value *i*. *i* is a three digit number with units corresponding to `title`, decades to `xlabel` and hundreds to `ylabel`. The actual meaning of options depends on the plot function (see `iiplot` for details). By adding a `c` after the command (`titoptc 111` for example), the choice is only applied to the current axis.

The title is built from the strings `LabLead` and `chlab` (for each displayed channel) which are initialized when the axis object is created. You can create your own title labels calls of the form `cf(`*i*`).LabLead` or `cf(`*i*`).chlab=`*CellArrayOfChannelLabels* with *i* the index of an existing axis of the `iiplot` or `feplot` figure with *SDT* `handle cf`.

`x [lin,log][ ,all], y [lin,log][ ,all],`

Default values for `xscale` *and* `yscale`. `xlin`, `xlog`, `ylin`, `ylog`, set values. `xy+1`, `xy+2` are used to toggle the `xscale` and `yscale` respectively (you can also use the **IIplot:xlin** and **IIplot:ylin** menus). Other commands are `xy1` for x-lin/y-lin, `xy2` for x-log/y-lin, `xy3` for x-lin/y-log, `xy4` for x-log/y-log.

You can all use the `all` qualifier to change all axes : `iicom('xlog all')`.

`wmin, wmax, w0, ...,`

Min/max frequency selection. Frequency bands are selected through the indices `IDopt(4:5)` for the first and last frequency.

`wmin` *f*	is used to find in `IIw` the index of the frequency point closest to *f*. `IDopt(4)` is then set to the value of this index. You can actually give minimum and maximum values using `wmin` *f1 f2*.
`wmax` *f*	sets `IDopt(5)` to the index of the frequency closest to *f*.
`wmo`	allows a mouse selection of the minimum and maximum frequency
`w0`	resets the values to cover the complete frequency range
`wnext`	shifts a reduced frequency window to the right (this is the action of the w button)

The toolbar shown above gives access to the commands `wmin`, `wnext`, `wmax`.

See also `iiplot`, section 3.2.3, `idcom`

iimouse

Purpose Mouse related callbacks for GUI figures.

Syntax

```
iimouse
iimouse('ModeName')
iimouse('ModeName',Handle)
```

Description `iimouse` is the general function used by `feplot` and `iiplot` to handle graphical inputs. While it is designed for *SDT* generated figures, `iimouse` can be used with any figure (make the figure active and type `iimouse`).

The main mouse mode is linked supports zooming and axis/object selection (see `zoom`). Context menus are associated to many objects and allow typical modifications of each object. When an axis is selected (when you pressed a button while your mouse was over it), `iimouse` reacts to a number of keys (see `key`). An active cursor mode (see `cursor`) has replaced the information part of previous versions of `iimouse`. 3-D orientation is handled by `view` commands.

`On,Off`

`iimouse` with no argument (which is the same as `iimouse('on')`) turn `zoom`, `key` and context `menu` on.

In detail, the `figure` is made `Interruptible`, `WindowButtonDownFcn` is set to `iimouse('zoom')` and `KeyPressFcn` to `iimouse('key')`).

Plot functions (`iiplot`, `feplot`) start `iimouse` automatically.

`iimouse off` turns all `iimouse` callbacks off.

`zoom`

This is basic mode of `iimouse`, it supports

- click and drag zoom into an area for both 2-D and 3-D plots (even when using perspective)
- zoom out to initial limits is obtained with a double click or the `i` key (on some systems the double click can be hard to control)

- active axis selection. `iimouse` makes the axis on which you clicked or the closest to where you clicked active (it becomes the current axis for `feplot` and `iiplot` figures).
- `colorbar` and `triax` axes automatically enter the `move` mode when made active
- `legend` axes are left alone but kept on top of other axes.

`cursor`

When you start the cursor mode (using the context menu opened with the right mouse button or by typing the `c` key), you obtain a red pointer that follows your mouse while displaying information about the object that you are pointing at. You can stop the cursor mode by clicking in the figure with your right mouse button or the `c` key. The object information area can be hidden by clicking on it with the right mouse button.

For `feplot` figures, additional information about the elements linked to the current point can be obtained in the MATLAB command window by clicking in the figure with the left button. By default, the cursor follows nodes of the first object in the `feplot` drawing axis. If you click on another object, the cursor starts pointing at it. In the wire-frame representation, particularly when using OpenGL rendering, it may be difficult to change object, the `n` key thus forces the cursor to point to the next object.

For `iiplot` axes, the cursor is a vertical line with circles on each data set and the information shows the associated data sets and values currently pointed at.

For `ii_mac` axes the current value of the MAC is shown.

`key`

Keyboard short-cuts. Some commands are directly available by pressing specific keys when a plot axis is active (to make it active just click on it with your mouse). The short cuts are activated by setting the `KeyPressFcn` to `iimouse('key')` (this is done by `iimouse on`). Short cuts are:

a,A	all axis shrink/expand	u,U	10^o horizontal rotation
c	start iimouse cursor	v,V	10^o vertical rotation
i	return to initial axis limits	w,W	10^o line of sight 10^o rotation
l,L	smaller/larger fecom scaledef	x,X	x/horizontal translation
n	cursor on next fecom object	y,Y	y/vertical translation
		z,Z	z/line of sight translation
-,_	previous (iicom ch-)	+,=	next (iicom ch+)
1,2,3,4	see view commands		

For feplot axes the translations are based on camera movements and correspond to the horizontal, vertical and line of sight for the current view. Translating along the line of sight has no effect without perspective and is similar to zooming with it. For other axes, the xyz keys translate along the data xyz directions.

move

The object that you decided to move (axes and text objects) follows your mouse until you click on a final desired position. This mode is used for triax (created by feplot) and colorbar axes, as well as text objects when you start move using the context menu (right button click to open this menu).

The moveaxis used for legend as a slightly different behavior. It typically moves the axis while you keep the button pressed.

You can call move yourself with `iimouse('move',Handle)` where Handle must be a valid axes or text object handle.

text

This series of commands supports the creation of a context menu for text objects which allows modification of font properties (it calls uisetfont), editing of the text string (it calls edtext), mouse change of the position (it calls iimouse), and deletion of the text object.

You can make your own text objects active by using `iimouse('textmenu',Handle)` where Handle must contain valid text object handle(s).

view,cv

iimouse supports interactive changes in the 3-D perspective of axes. Object views are controlled using azimuth and elevation (which control the orientation vector linking the CameraTarget and the CameraPosition) and self rotation (which control the CameraUpVector). You can directly modify the view of the current axis using the Matlab view and cameramenu functions but additional capabilities and automated orientation of triax axes are supported by iimouse.

1	first standard view. Default n+y. Changed using the View default context menu.
2	standard xy view (n+z). Similar to Matlab view(2) with resetting of CameraUpVector. Changed using the View default context menu.
3	standard view. Default to Matlab view(3).
4	standard view. Default n+x.
n[+,-][x,y,z]	2-D views defined by the direction of the camera from target.
n[+,-][+,-][+,-]	3-D views defined by the signs projection of line of sight verctor along the xyz axes.
dn ...	dn commands allow setting of default 1234 views. Thus viewdn-x will set the 4 view to a normal along negative x
az el sr	specify azimuth, elevation and rotation around line of sight
g *rz ry rz*	specify rotations around global *xyz* axes
[x,y,z][+,-] *ang*	rotation increments around global *xyz* axes
[h,v,s][+,-] *ang*	current horizontal, vertical and line of sight axes

All angles should be specified in degrees.

iimouse key supports rotations by +/- 10 degrees around the current horizontal, vertical and line of sight axes when any of the u, U, v, V, w, W keys are pressed (same as fecom('viewh-10') ...). 1, 2, 3, 4 return to basic 2-D and 3-D views.

iimouse('cv') returns current view information you can then set other axes with iimouse('view',AxesHandles,cv).

See also iicom, fecom, iiplot, iiplot

iiplot

Purpose Refresh all the drawing `axes` of the `iiplot` interface.

Syntax

```
iiplot
```

Description `iiplot` is used as a command with no arguments is the user entry point to refresh all the drawing axes. Section 3.2 gives an introduction to the use of `iiplot` and the companion function `iicom`. `cf=iiplot` returns a *SDT* `handle` to the current `iiplot` figure. You can create more than one `iiplot` figure with `cf=iiplot(`*`FigHandle`*`)`.

`iiplot` axes take their data in the database wrapper object stored in the standard global variable `XF`. In the nominal configuration, the contents of `XF` are

```
XF (global variable) = Database Wrapper (SDT Handle object)

{1} [.w  (IIw)   1024x1, .xf  (IIxf)   1024x33] : response data
 2  [.w  (IIw)   1024x1, .xf  (IIxe)   1024x33] : response data
 3  [.w  (IIw)   1024x1, .xf  (IIxh)       0x0] : response data
 4  [.w  (IIw)   1024x1, .xf  (IIxi)       0x0] : response data
 5  [.po (IIpo)     0x0, .res (IIres)      0x0] : shape data
 6  [.po (IIpo1)    0x0, .res (IIres1)     0x0] : shape data
```

where the interface uses standard global variables `IIw` (frequencies), `IIxf` (main data set), `IIxe` (identified model), `IIxh`, `IIxi` (alternate data sets), `IIpo`, `IIpo1` (main/alternate pole sets), `IIres`, `IIres1` (main/alternate residue sets). **Note** that two other global variables `XFdof` (which describes DOFs at which the responses/shapes are defined, see `.dof` field for response and shape data in the `xfopt` section) and `IDopt` (which contains options used by identification routines, see `idopt`) are also pointed at by `XF`.

Each `iiplot` axis can display some or all these data sets. The selection of what is displayed is obtained using the `iicom IIxf`, `IIxfOn`, ... commands or the `Variables` menu.

The frequencies associated with `XF(1)` are always stored in `IIw` and frequency band selection with the `w` commands always refer to `IIw`. You can use different frequency values for the other data sets by setting `XF(2).w` ... to something different than `IIw`.

Plot types supported by `iiplot` are described below. The plot type can be selected using the `PlotType` menu of the toolbar or through `iicom show` commands.

Selected channels (columns of the data sets) are shown for all plots. The `iicom` commands `+`, `-`, `ch`, `ad` and the associated keys and toolbar buttons can be used to change selected channels.

Pole lines for the indication of pole frequencies are available for many plots. The `iicom PoleLine` commands and the `IIplot:PoleLine` menu can be used to change how these lines appear. `IIpo` pole lines are shown in white/black. `IIpo1` pole lines in red.

Default scale type. `0` x-lin/y-lin, `2` x-log/y-lin, `3` x-lin/y-log, and `4` x-log/y-log. Default scale types can be selected using the `IIplot:xlin` and `IIplot:ylin` menus of the *command figure*, or through the `iicom xy` commands.

Automated title/label generation options are changed using the `iicom('titopt i')`. Where *i* is a three digit number with units corresponding to `title`, decades to `xlabel` and hundreds to `ylabel`. Use the `iiplot:TitOpt` menu to see available options.

For example the default is `iicom('titopt 111')` which for an amplitude plot shows `ylabel('Amplitude (m/N)')` (label and units), `xlabel('Frequency (Hz)')` (label and units), and `title('a 1')` (Address number). Unit labels can be changed through the database wrapper object `XF` (see `xfopt`). Modifying the `IDopt unit` value changes the unit assumed for identification but not the dataset units. Address labels used for titles can be set using the `xfopt DofLabel` command.

Titles and labels are not regenerated when using `ch` commands. If something is not up to date, use `iicom sub` which rechecks everything.

`Abs,Pha,Phu,Rea,Ima,R&I,Nyq`

Basic plots are *amplitude of response* (initialized by `iicom('show abs')`), *phase of response* wrapped (`show pha`) or unwrapped (`show phu`), *real part of the response* (`show rea`), *imaginary part* (`show ima`), *real and imaginary parts* (`show r&i`), or *Nyquist plot*(real versus imaginary part, `show nyq`).

Title options, data set, channel and frequency band selection are described above.

`Local Nyquist`

Local Nyquist plots (initialized by `show lny`) show a comparison of `IIxf` (measured FRFs) and `IIxe` (identified model) in a reduced frequency band

$$\left[\begin{array}{cc} \omega_j(1-\zeta_j) & \omega_j(1+\zeta_j) \end{array}\right]$$

near the currently selected pole (the current pole is selected by clicking on a pole line in another plot axis). Local Nyquist plots allow a local evaluation of the quality of the fit. The `error` and `quality` plots give a summary of the same information for all the frequency response functions and all poles.

`MMIF, MMIF forces, AMIF, SUM, CMIF, SumI`

Multivariate Mode Indicator Function (initialized by `show mmi`), *forces associated to the MMIF* (initialized by `show fmi`), `Alternate Mode Indicator Function` (`show ami`), and *Channel Sum* (`show sum`) are four ways to combine all the FRFs or a set to get an indication of where its poles are located.

These indicators are discussed in the `ii_mmif` *Reference* section. They are automatically computed by `iiplot` based on data in the `XF(1)` set (which normally corresponds to data in the `IIxf` global variable).

`Pole, Freq/damp`

Pole locations in the complex plane (initialized by `show pol`).

Poles shown as damping vs. frequency are initialized by `show fre`.

`Residues`

Position of residues in the complex plane are initialized by `show cre`. This plots can be used to visualize the phase scatter of identified residues.

Value of real residue for each measured channel are initialized by `show rre`.

`Error, Quality`

Local Nyquist error (initialized by `show err`). For the current pole, error plots select frequency points in the range $[\omega_j(1-\zeta_j) \;\; \omega_j(1+\zeta_j)]$. For each channel (FRF column), the normalized error (RMS response of `IIxe-IIxf` divided by RMS response of `IIxf`) is shown as a dashed line with `+` markers and a normalized response level (RMS response of `IIxf`) as a dashed line with `x` markers.

Normalized errors should be below 0.1 unless the response is small.

Mode quality plot (initialized by `show qua`), gives a mean of the local Nyquist plot. The dashed lines with `+` and `x` markers give a standard and amplitude weighted mean of the normalized error. The dotted line gives an indication of the mean response level (to see if a mode is well excited in the FRFs). Normalized errors should be below 0.1 unless the response is small.

See also `iicom`, `iiplot`, `setlines`, `xfopt`

ii_cost

Purpose Compute the quadratic and log-least-squares cost functions comparing two sets of frequency response functions.

Syntax `[cst] = ii_cost(xf,xe)`

Description For two sets of FRFs H and $\hat{H}$, the quadratic cost function is given by

$$J_{ij}(\Omega) = \sum_{ij\ \text{measured}, k\in\Omega} |\hat{H}_{ij}(s_k) - H_{ij}(s_k)|^2$$

and the log-least-square cost function by

$$J_{ij}(\Omega) = \sum_{ij\ \text{measured}, k\in\Omega} |log\left|\frac{\hat{H}_{ij}(s_k)}{H_{ij}(s_k)}\right||^2$$

For sets `xf` and `xe` stored using the `xf` format (see page 40), `ii_cost` computes both those costs which are used in identification and model update algorithms (see section 4.2.3).

See also `id_rc`, `up_ixf`

ii_mac

Purpose User interface function for MAC and other vector correlation criteria.

Syntax

```
             ii_mac(cpa,cpb)
VC         = ii_mac(cpa,cpb,'PropertyName',PropertyValue, ...)
[VC,ReS] = ii_mac('PropertyName',PropertyValue, ... ,'Command')
             ii_mac(Fig,'PropertyName',PropertyValue, ... ,'Command')
Result =     ii_mac(Fig ,'Command')
VC.PropertyName = PropertyValue
```

Description

The `ii_mac` function gives access to vector correlation tools provided by the *SDT* starting with the Modal Assurance Criterion (MAC) but including many others. A summary of typical applications is given in section 4.2 and examples in the `gartco` demo.

Vector correlations are *SDT* objects which contain at least two sets of deformations `cpa` and `cpb` (the names stand for $[c]\{\phi_a\}$ and $[c]\{\phi_b\}$ since these vectors typically represent the observation of modeshapes at test sensors, see section 2.1).

The details about possible fields of vector correlation objects are given after the listing of supported commands below.

GUI

If you use `ii_mac` without requesting graphical output, the vector correlation object is deleted upon exit from `ii_mac`. In other cases, the object is saved in the figure so that you can reuse it.

The `II_MAC menu` lets you choose from commands that can be computed based on the data that you have already provided. The `context menu` associated with plots generated by `ii_mac` lets you start the cursor, display tabular output, ...

You can add data to other fields or call new commands from the command line by starting the `ii_mac` call with a pointer to the figure where the vector correlation is stored (`ii_mac(fig,'Command')`, ...). An alternate calling form is to set a field of the vector correlation object.

The following commands

```
load sdt_gart;
model=struct('Node',FEnode,'Elt',FEelt,'DOF',mdof,'pl',pl,'il',il);
sens=fe_sens('model',model);sens.tdof=sdof;
```

```
sens=fe_sens('arigid',sens);
[m,k,mdof] = fe_mk(model,'options',[0 1]);

figure(1); subplot(221);
VC=ii_mac(IIres.',md0,'labela','Test','labelb','FEM', ...
          'sens',sens,'Mac Pair Plot');
subplot(212);ii_mac(1,'comac');
VC.m = m; VC.kd = ofact(k+1e1*m);
subplot(222); VC.MacMPairPlot;
```

illustrate a fairly complex case where one shows the MAC in `subplot(221)`, all three COMAC indicators in `subplot(212)`, then provide mass and a mass-shifted stiffness to allow computation of the mass condensed on sensors and finally show the reduced mass weighted MAC in `subplot(222)`.

Commands

`COMAC [ ,M][,A,B][,N][,S][,E] [,sort][,table,tex]`

The `COMAC` command supports three correlation criteria (nominal, scaled and enhanced) whose objective is to detect sensors that lead to poor correlation. You can compute all or some of these criteria using the `n`, `s`, or `e` modifiers (with no modifier the command computes all three).

The output is either graphical or tabulated (`table` and `tex` modifiers). Sensors are given in the nominal order or sorted by decreasing COMAC value (`sort` modifier).

These criteria assume the availability of paired sets of sensors. The `COMAC` commands thus start by using `MacPair` (`MacMPair` with the `M` modifier) to pair vectors in `cpb` to vectors in `cpa`. The `B` modifier can be used to force pairing against vectors in set B (rather than A which is the default value).

The **nominal** Coordinate Modal Assurance Criterion (COMAC) measures the correlation of two sets of similarly scaled modeshapes at the same sensors. The definition used for the *SDT* is

$$\text{COMAC}_l = 1 - \frac{\left\{\sum_j^{NM} |c_l\phi_{jA} c_l\phi_{jB}|\right\}^2}{\sum_j^{NM} |c_l\phi_{jA}|^2 \sum_j^{NM} |c_l\phi_{jB}|^2} \tag{9.4}$$

which is 1 minus the definition found in [48] in order to have good correlation correspond to low COMAC values.

The assumption that modes a similarly scaled is sometimes difficult to ensure, so that the **scaled** COMAC is computed with shapes in set B scaled using the Modal Scale Factor (MSF)

$$\left\{\widehat{c\phi_{jB}}\right\} = \{c\phi_{jB}\}\,\mathrm{MSF}_j = \{c\phi_{jB}\}\,\frac{\{c\phi_{jB}\}^T\,\{c\phi_{jA}\}}{\{c\phi_{jB}\}^T\,\{c\phi_{jB}\}} \tag{9.5}$$

which sets the scaling of vectors in set B to minimize the quadratic norm of the difference between $\{c\phi_{jA}\}$ and $\left\{\widehat{c\phi_{jB}}\right\}$.

The **enhanced** COMAC (eCOMAC), introduced in [49], is given by

$$\mathrm{eCOMAC}_l = \frac{\sum_j^{NM}\left\|\left\{\widetilde{c_l\phi_{jA}}\right\} - \left\{\widehat{c\phi_{jB}}\right\}\right\|}{2NM} \tag{9.6}$$

where the comparison is done using modeshapes that are vector normalized to 1

$$\left\{\widetilde{c_l\phi_{jA}}\right\} = \{c\phi_{jA}\}\,/\,\|c\phi_{jA}\|$$

This is an example of how to use of the COMAC command

```
[model,sens,ID,FEM]=demosdt('demopairmac');

figure(1);clf;
ii_mac(ID,FEM,'sens',sens,'comac plot')
ii_mac(1,'comac table');
```

MAC [,PairA,PairB][Plot,Table,Tex,Thtml]

The Modal Assurance Criterion (MAC) [11] is the most widely used criterion for vector correlation (mainly because of its simplicity).

The MAC is the correlation coefficient of vector pairs in two vector sets `cpa` and `cpb` defined at the same DOFs. In general `cpa` corresponds to measured modeshapes at a number of sensors $\{c\phi_{\mathtt{id}j}\}$ while `cpb` corresponds to the observation of analytical modeshapes $[c]\{\phi_k\}$. The MAC is given by

$$\mathrm{MAC}_{jk} \quad = \quad \frac{|\,\{c\phi_{\mathtt{id}j}\}^H\ \ \{c\phi_k\}\,|^2}{|\,\{c\phi_{\mathtt{id}j}\}^H\ \ \{c\phi_{\mathtt{id}j}\}\,|\,|\,\{c\phi_k\}^H\ \ \{c\phi_k\}\,|} \tag{9.7}$$

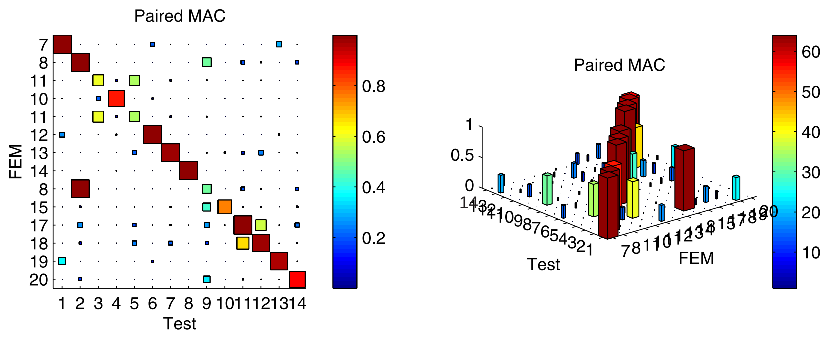
For two vectors that are proportional the MAC equals 1 (perfect correlation). Values above 0.9 are generally considered as very correlated. Values below 0.6 should be considered with much caution (they may or may not indicate correlation).

The figure below shows the standard 2-D (obtained using the context menu or `view(2)`) and 3-D (obtained using the context menu or `view(-130,20)`) representations of the MAC supported by `ii_mac`. The color and dimensions of the patches associated to each vector pair are proportional to the MAC value.

The basic MAC shows vector pairs for all vectors in the two sets. The `MacPair` command seeks for each vector in `cpa` (`cpb` with `PairB`) the vector in `cpb` (`cpa`) that is best correlated. The result only shows vectors that are best correlated.

By default (or with `MacPlot`), the command plots the result as shown below. You can obtain tabulated text output using `Mac Table` which can be pasted into Word and transformed into a table. `Mac Tex` gives a format suitable for direct inclusion in LaTeX. `Mac Thtml` creates and gives a HTML file.

If `cpa` is defined at sensors and `cpb` at DOFs, `ii_mac` uses the sensor configuration `sens` to observe the motion of `cpb` at sensors. If `cpa` is defined at DOFs and `cpb` at sensors, `ii_mac` calls `fe_exp` to expand `cpb` on all DOFs. For vectors defined at all DOFs, the MAC is a poor criterion and you should really use its mass weighted counter part.



This is an example of how to use of the MAC command

```
[model,sens,ID,FEM]=demosdt('demopairmac');

figure(1);clf;
ii_mac(ID,FEM,'sens',sens,'mac paira plot')
ii_mac(1,'mac paira table');
```

A few things you should know ...

The MAC measures the shape correlation without any reference to scaling of each vector (because of the denominator in (9.7)). This makes the MAC easy to use but also limits its applicability (since the modeshape scaling governs the influence of a given mode on the overall system response, a proper scaling is necessary when comparing the relative influence of different modes). In other terms, the MAC is not a norm (two vectors can be very correlated and yet different), so care must be taken in interpreting results.

As the MAC is insensitive to mode scaling, it can be used with identified normal mode residues. Thus for `RealRes` found with `id_rc` and `cphi` resulting from a FE analysis, `ii_mac(RealRes(1:np,:)',cphi)` directly gives a measure of the test/analysis correlation.

The main weakness of the MAC is linked to scaling of individual components in the correlation. A change in sensor calibration can significantly modify the MAC. If the nature of various sensors are different (velocity, acceleration, deformation, different calibration, ...) this can induce significant problems in interpretation.

The reference weighting in mechanics is energy. For incomplete measurements, kinetic energy can be approximated using a static condensation of the mass on the chosen sensors which can be computed using the `MacM` command.

`MAC Auto[A,B][Plot,Table,Tex,Thtml]`

Since the objective of the MAC is to estimate the correlation between various vectors, it is poor practice to have vectors known to be different be strongly correlated.

Strong correlation of physically different vectors is an indication of poor test design or poor choice of weighting. `MacAutoA` (`B`) compute the correlation of `cpa` (`cpb`) with itself. Off diagonal terms should be small (smaller than 0.1 is generally accepted as good practice).

On certain systems where the density of sensors is low on certain parts, cross-correlation levels with the mass weighted MAC can be much lower than for the unweighted MAC. In such cases, you should really prefer the mass weighted MAC for correlation.

This is an example of how to use of the MACAuto command

```
[model,sens,ID,FEM]=demosdt('demopairmac');

figure(1);clf;
subplot(1,2,1);ii_mac(ID,FEM,'sens',sens,'mac autoa plot')
subplot(1,2,2);ii_mac(ID,FEM,'sens',sens,'mac autob plot')
```

```
ii_mac(1,'mac autoa table');ii_mac(1,'mac autob table');
```

`MACCo [ ,M] [,`*ns*`]`

The `MACCo` criterion is a *what if* analysis. It takes modes in `cpa`, `cpb` and computes the paired MAC or MAC-M with one sensor removed. The sensor removal leading to the best mean MAC for the paired modes is a direct indication of where the poorest correlation is found. The algorithm removes this first sensor then iteratively proceeds to remove *ns* other sensors (the default is 3). The `MACCo` command prints an output of the form

```
Sensor Mean   1   2   3   4   5   6   7   8   9  10  11  12  13  14
Initial   86 100  98  60  86  53  99  98 100  48  76  99  97  96  88
1302-y    87 100  99  61  86  54  99  98 100  60  76 100  97  96  89
1112z     87 100  99  61  88  62  99  98 100  59  76 100  98  95  89
1005z     88 100  99  63  88  62  99  98 100  65  79 100  98  95  89
```

where the initial mean MAC and MAC associated to the paired vectors are shown in the first line, and the evolution of these quantities when sensors are removed. While the selection criteria is the mean of the MAC for all the paired modes, the individual MAC are displayed to give an indication of how individual modes evolve. Here for example, sensor `1302-y` mostly has a negative impact on the correlation of mode 9.

The sensor labels are replaced by sensor numbers if the sensor configuration `sens` is not declared.

This is an example of how to use of the MACCo command

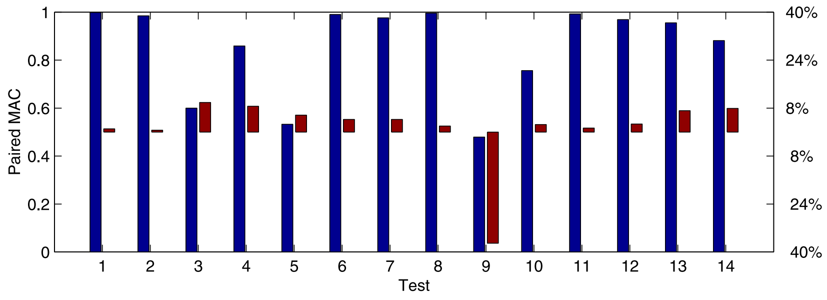
```
[model,sens,ID,FEM]=demosdt('demopairmac');
ii_mac(ID,FEM,'sens',sens,'macco')
```

`MAC [ ,M] Error [,Table,Tex,Thtml]`

Computes the MAC (or MAC-M), does pairing and plots a summary combining the MAC value found for paired modes and the associated error on frequencies (`(fb-fa)./fa`). A typical call (see `gartco` for example) would be

```
ii_mac('cpa',IIres.','cpb',md0,'sens',sens, ...
 'fa',IIpo,'fb',f0,'labela','Test','labelb','FEM', ...
 'mac error plot');
```

By default this command displays a plot similar to the one shown below where the diagonal of the paired MAC and the correspond relative error on frequencies are shown. With the `table` modifier, `ii_mac` gives the result as a table in the MATLAB command window.



This is an example of how to use of the MACError command

```
[model,sens,ID,FEM]=demosdt('demopairmac');
ii_mac(ID,FEM,'sens',sens,'macerror plot')
ii_mac(ID,FEM,'sens',sens,'macerror table')
```

`Mac M ...`

When `cpa` and `cpb` are defined at finite element DOFs, it is much more appropriate to use a mass weighted form of the MAC defined as

$$\text{MAC-M}_{jk} = \frac{|\left\{\phi_{jA}\right\}^T [M] \left\{\phi_{kB}\right\}|^2}{|\left\{\phi_{jA}\right\}^T [M] \left\{\phi_{jA}\right\}| |\left\{\phi_{kB}\right\}^T [M] \left\{\phi_{kB}\right\}|} \tag{9.8}$$

When `cpa` and `cpb` are defined at sensors using a mass weighting is always a good idea. If vectors are defined as sensors, the problem is to define what the mass should be. The standard approach is to use the static condensation of the full order model mass on the sensor set. The way this reduced mass is computed in the *SDT* is discussed in the `MC` section below.

If `cpa` is defined at sensors and `cpb` at DOFs, `ii_mac` uses the sensor configuration `sens` to observe the motion of `cpb` at sensors. If `cpa` is defined at DOFs and `cpb` at sensors, `ii_mac` calls `fe_exp` to expand `cpb` on all DOFs.

The MAC-M can be seen as a scale insensitive version of the Pseudo-Orthogonality check (also called Cross Generalized Mass criterion) described below.

The PairA, PairB, AutoA, AutoB, Plot, Table modifiers are available for MacM just as for the MAC. A short call format is `ii_mac(cpa,cpb,m,'mac m plot')` where cpa, cpb and m are given as the first three input arguments.

`POC [,Pair[A,B]][Plot,Table,Tex,Thtml]`

The orthogonality conditions (6.5) lead to a number of standard vector correlation criteria. The **pseudo-orthogonality check** (POC) (also called **Cross generalized mass** (CGM)) and the less commonly used cross generalized stiffness (CGK) are computed using

$$\mu_{jk} = \{\phi_{jA}\}^T [M] \{\phi_{kB}\} \qquad \kappa_{jk} = \{\phi_{jA}\}^T [K] \{\phi_{kB}\} \tag{9.9}$$

where for mass normalized test and analysis modes one expects to have $\mu_{jk} \approx \delta_{jk}$ and $\kappa_{jk} \approx \omega_j^2 \delta_{jk}$.

For matched modes, POC values differing significantly from 1 indicate either poor scaling or poor correlation. To distinguish between the two effects, you can use a MAC-M which corresponds to the square of a POC where each vector would be normalized first (see the MacM command).

Between unmatched modes, POC values should be close to zero. In some industries, off-diagonal cross POC values below 0.1 are required for the test verification of a model.

The PairA, PairB, Plot, Table modifiers are available for POC just as for the MAC.

`Rel [,scaled][,m]`

For scaled matched modeshapes, the **relative error**

$$e_j = \frac{\| \{c\phi_{jA}\} - \{c\phi_{jB}\} \|}{\| \{c\phi_{jA}\} \| + \| \{c\phi_{jB}\} \|} \tag{9.10}$$

is one of the most accurate criteria. In particular, it is only zero if the modeshapes are exactly identical and values below 0.1 denote very good agreement.

The rel command calls MacPair to obtain shape pairs and plots the result of (9.10).

For unscaled matched modeshapes, you may want to seek for each vector in set B a scaling coefficient that will minimize the relative error norm. This coefficient is known as the **modal scale factor** and defined by

$$\text{MSF}_j = \frac{\{c\phi_{jA}\}^T \{c\phi_{jB}\}}{\{c\phi_{jB}\}^T \{c\phi_{jB}\}} \tag{9.11}$$

The `RelScale` command calls `MacPair` to obtain shape pairs, multiplies shapes in set B by the modal scale factor and plots the result of (9.10).

With the `M` modifier, the `MacPairM` is used to obtain shape pairs, kinetic energy norms are used in equations (9.10)-(9.11).

This is an example of how to use of the Rel command

```
[model,sens,ID,FEM]=demosdt('demopairmac');
ii_mac(ID,FEM,'sens',sens,'rel');
```

Fields The following sections describe standard fields of vector correlation objects and how they can be set.

`VC.va`	vector set A detailed below
`VC.vb`	vector set B detailed below.
`VC.sens`	sensor description array describing the relation between the DOFs of `cpb` and the sensors on which `cpa` is defined.
`VC.m`	full order mass matrix
`VC.mc`	reduced mass matrix defined at sensors (see definition below)
`VC.qi`	sensor confidence weighting
`VC.k`	full order stiffness matrix
`VC.kd`	factored stiffness or mass shifted stiffness matrix
`VC.T`	Reduced basis used for dynamic expansion

`va`

Vector set A, associated frequencies or poles and label. For identification results, you would typically use

```
ii_mac(FigHandle,'cpa',IIres.','fa',IIpo,'labela','Test');
```

You can also use standard data structures used in identification. Pole/residue models with `.res` and `.po` fields (see section 2.6) or frequency responses with `.w` and `.xf` fields (see section 2.8).

If `cpa` is defined at sensors and `cpb` at DOFs, `ii_mac` uses the sensor configuration `sens` to observe the motion of `cpb` at sensors. If `cpa` is defined at DOFs and `cpb` at sensors, `ii_mac` calls `fe_exp` to expand `cpb` on all DOFs.

`vb`

Vector set B, associated frequencies or poles and label. If this set represents finite element vectors, you will typically declare it as

```
ii_mac(FigHandle,'cpb',mode,'fa',freq,'labela','FEM','sens',sens);
```

where the sensor configuration description `sens` is obtained using `fe_sens` (see the `gartte` demo for an example).

`m,k,kd`

For criteria that use vectors defined at DOFs, you may need to declare the mass and stiffness matrices. For large models, the factorization of the stiffness matrix is particularly time consuming. If you have already factored the matrix (when calling `fe_eig` for example), you should retain the result and declare it in the `kd` field.

The default value for this field is `kd=ofact(k,'de')` which is not appropriate for structures with rigid body modes. You should then use a mass-shift (`kd = ofact( k + alpha*m,'de')`, see section 6.1.4).

`mc`

The *SDT* supports an original method for reducing the mass on the sensor set. Since general test setups can be represented by an observation equation (4.1), the principle of reciprocity tells that $[c]^T$ corresponds to a set of loads at the location and in the direction of the considered sensors. To obtain a static reduction of the model on the sensors, one projects the mass (computes T^TMT) on the subspace

$$[T] = \left[\tilde{T}\right]\left[c\tilde{T}\right]^{-1} \quad \text{with} \quad [K]\left[\tilde{T}\right] = [c]^T \tag{9.12}$$

In cases where the model is fixed $[K]$ is non-singular and this definition is strictly equivalent to static/Guyan condensation of the mass [18]. When the structure is free, $[K]$ should be replaced by a mass shifted $[K]$ as discussed under the `kd` field above.

`sens`

Sensor configuration description. This data structure is initialized with `fe_sens` and contains a description of how the FE model and test configuration are related.

FEM results are always assumed to be placed in the `.vb` field. One thus compares `VC.va.def` and `VC.sens.cta*VC.vb.def`.

`T`

Reduced basis expansion methods were introduced in [18]. Static expansion can be obtained by using `T` defined by (9.12).

To work with dynamic or minimum residual expansion methods, `T` should combine static shapes, low frequency modes of the model, the associated modeshape sensitivities when performing model updating.

Modeshape expansion is used by `ii_mac` when `cpa` is full order and `cpb` is reduced. This capability is not currently finalized and will require user setting of options. Look at the HTML or PDF help for the latest information.

See also `ii_comac`, `fe_exp`, the `gartco` demonstration, section 4.2

ii_mmif

Purpose Compute various Mode Indicator Functions

Syntax

```
[mmif,ua] = ii_mmif(xf,IDopt)
[mmif,ua] = ii_mmif(xf,IDopt,'waitbar')
amif =  ii_mmif(xf,IDopt,'amif','waitbar')
[cmif,u,v] = ii_mmif(xf,IDopt,'cmif','waitbar')
sum = ii_mmif(xf,IDopt,'sum')
```

Description Mode indicator functions seek to combine data from several input/output pairs of a MIMO transfer function in a single response that gives the user a visual indication of pole locations. You can then use the `idcom e` command to get a pole estimate.

This function supports standard mode indicator functions and is easily accessed through `iiplot` interface which computes and displays these functions for the nominal data set (`XF(1)`).

MMIF The Multivariate Mode Indicator Function (MMIF) (use the `iicom showmmi` command) was introduced in [50]. It's introduction is motivated by the fact that, for a single mode mechanical model, the phase at resonance is close to -90^o. For a set of transfer functions such that $\{y(s)\} = [H(s)]\{u(s)\}$, one thus considers the ratio of real part of the response to total response

$$q(s,\{u\}) = \frac{\{u\}^T\left[\mathrm{Re}H^T\mathrm{Re}H\right]\{u\}}{\{u\}^T\left[H^HH\right]\{u\}} = \frac{\{u\}^T[B]\{u\}}{\{u\}^T[A]\{u\}} \tag{9.13}$$

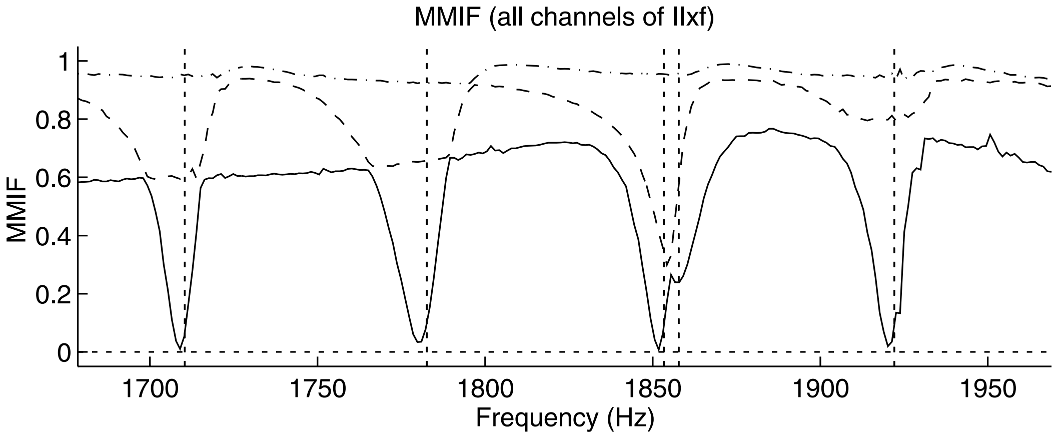
For structures that are mostly elastic (with low damping), resonances are sharp and have properties similar to those of isolated modes. The MMIF (q) thus drops to zero.

Note that the real part is considered for force to displacement or acceleration, while for force to velocity the numerator is replaced by the norm of the imaginary part in order to maintain the property that resonances are associated to minima of the MMIF. A MMIF showing maxima indicates improper setting of `IDopt.DataType`.

For system with more than one input (u is a vector rather than a scalar), one uses the extrema of q for all possible real valued u which are given by the solutions of the eigenvalue problem $[A]\{u\}q + [B]\{u\} = 0$.

The figure below shows a particular set fo MMIF. The system has 3 inputs, so that there are 3 indicator functions. The resonances are clearly indicated by minima that are close to zero.

The second indicator function is particularly interesting to verify pole multiplicity. It presents an minima when the system presents two closely spaced modes that are excited differently by the two inputs (this is the case near 1850 Hz in the figure). In this particular case, the two poles are sufficiently close to allow identification with a single pole with a modeshape multiplicity of 2 (see id_rm) or two close modes. More details about this example are given in [13].



This particular structure is not simply elastic (the FRFs combine elastic properties and sensor/actuator dynamics linked to piezoelectric patches used for the measurement). This is clearly visible by the fact that the first MIF does not go up to 1 between resonances (which does not happen for elastic structures).

At minima, the forces associated to the MMIF (eigenvector of $[A]\{u\}q+[B]\{u\}=0$) tend to excite a single mode and are thus good candidates for force appropriation of this mode [51]. These forces are the second optional output argument ua.

CMIF The Complex Mode Indicator Function (CMIF) (use the iicom showcmi command, see [52] for a thorough discussion of CMIF uses), uses the fact that resonances of lightly damped systems mostly depend on a single pole. By computing, at each frequency point, the singular value decomposition of the response

$$[H(s)]_{NS\times NA}=[U]_{NS\times NS}\,[\Sigma]_{NS\times NA}\left[V^H\right]_{NA\times NA} \tag{9.14}$$

one can pick the resonances of Σ and use U_1,V_1 as estimates of modal observability / controllability (modeshape / participation factor). The optional `u`, `v` outputs store the left/right singular vectors associated to each frequency point.

AMIF `ii_mmif` provides an alternate mode indicator function defined by

$$q(s) = 1 - \frac{|\text{Im}H(s)||H(s)|^T}{|H(s)||H(s)|^T} \tag{9.15}$$

which has been historically used in force appropriation studies [51]. Its properties are similar to those of the MMIF except for the fact that it is not formulated for multiple inputs.

This criterion is supported by `iiplot` (use `iicom('show ami)`).

SUM Thus sum of the amplitude of all channels

$$S(s) = \sum_{j,k} \|H_{j,k}(s)\|$$

is another function sometimes used as a mode indicator function and is thus supported by `ii_mmif` (use `iicom('show sum)`).

SUMI Thus sum of the square of the imaginary part of all channels

$$S(s) = \sum_{j,k} \text{Im}(H_{j,k}(s))^2$$

is another function sometimes used as a mode indicator function and is thus supported by `ii_mmif` (use `iicom('show sumi)`).

See also `iiplot`, `iicom`, `idopt`, `fe_sens`

ii_plp

Purpose Pole frequency indication using vertical lines.

Syntax

```
ii_plp(po)
ii_plp(po,color,Opt)
```

Description `ii_plp(po)` will overplot dotted vertical lines indicating the pole frequencies of complex poles in `po` and dashed lines at the frequencies of real poles. The poles `po` can be specified in any of the 3 accepted formats (see `ii_pof`).

When you click on these lines, a text object indicating the properties of the current pole is created. You can delete this object by clicking on it. When the lines are part of `iiplot` axes, clicking on a pole line changes the current pole and updates any axis that is associated to a pole number (local Nyquist, residue and error plots, see `iiplot`).

The optional `color` argument can be used to obtain something else than white/black. In the `iiplot` interface for example, frequencies of poles in the alternate pole set `IIpo1` are shown in red.

Other options are given in the cell array `Opt={Unit,ForMat,VariableName}`.

The integer `Units` with tens set to `1` (`11` or `12`) is used for poles in Hz, while those with tens set to `2` correspond to Rad/s. This value is typically obtained from `IDopt(3)`.

The integer `Format` specifies whether the imaginary part $Im(\lambda)$ (`Format=2` which is the default) or the amplitude $|\lambda|$ (using `Format=3` corresponding to format 3 of `ii_pof`) should be used as the "frequency" value for complex poles.

`VariableName` can be used to pass the name of the pole variable (this is to create the info string obtained when you click on the pole line).

See also `ii_pof`, `idopt`, `iiplot`, `iicom`

ii_poest

Purpose Identification of a narrow-band single pole model.

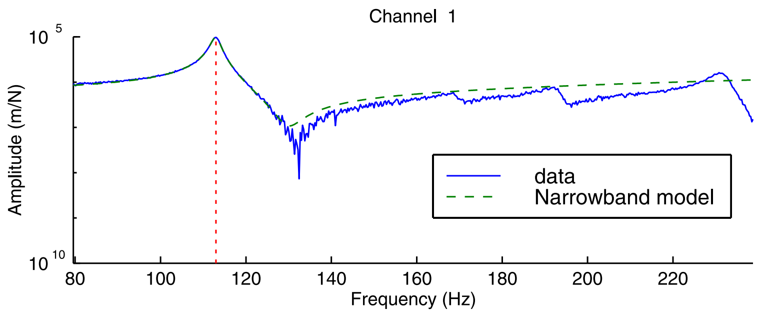
Syntax

```
po          = ii_poest(xf,w,idopt)
[res,po,xe] = ii_poest(xf,w,idopt,opt)
```

Description ii_poest (idcom e command and associated button in the idcom GUI figure, see section 3.4) provides local curve fitting capabilities to find initial estimates of poles by simply giving an indication of their frequency.

The central frequency for the local fit is given as opt(2) or, if opt(2)==0, by clicking on a plot whose abscissas are frequencies (typically FRF of MMIF plots generated by iiplot).

The width of the selected frequency band can be given in number of points (opt(1) larger than 1) or as a fraction of the central frequency (points selected are in the interval opt(2)*(1+[-opt(1) opt(1)]) for opt(1)<1). The default value is opt(1)=0.01.



A single pole fit of the FRFs in xf is determined using a polynomial fit followed by an optimization using a special version of the id_rc algorithm. The accuracy of the results can be judged graphically (when using the idcom e command, IIxf and IIxe are automatically overlaid as shown in the plot above) and based on the message passed

```
LinLS:  1.563e-11, LogLS  8.974e-05, nw 10
mean(relE) 0.00, scatter 0.00
Found pole at 1.1299e+02   9.9994e-03
```

which indicates the linear and quadratic costs (see `ii_cost`) in the narrow frequency band used to find the pole, the number of points in the band, the mean relative error (norm of difference between test and model over norm of response, see `iiplot error`) which should be below 0.1, and the level of scatter (norm of real part over norm of residues, which should be small if the structure is close to having proportional damping).

If you have a good fit and the pole differs from poles already in your current model, you can add the estimated pole (add `IIpo1` to `IIpo`) using the `idcom ea` command.

The choice of the bandwidth can have a significant influence on the quality of the identification. As a rule the bandwidth of your narrow-band identification should be larger than the pole damping ratio (`opt(1)=0.01` for a damping of 1% is usually efficient). If, given the frequency resolution and the damping of the considered pole, the default does not correspond to a frequency band close to $2\zeta_j\omega_j$, you should change the selected bandwidth (for example impose the use of a larger band with `opt(1)=.02` which you can obtain simply using `idcom ('e.02')`).

This routine should be used to obtain an initial estimate of the pole set, but the quality of its results should not lead you to skip the pole tuning phase (`idcom eup` or `eopt` commands) which is essential particularly if you have closely spaced modes.

See also `idcom`, `id_rc`, `iiplot`

ii_pof

Purpose Transformations between the three accepted pole formats.

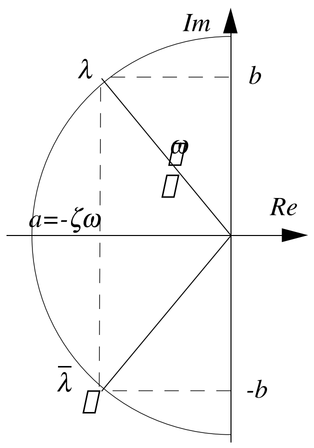
Syntax

```
[pob] = ii_pof(poa,DesiredFormatNumber)
[pob] = ii_pof(poa,DesiredFormatNumber,SortFlag)
```

Description The *Structural Dynamics Toolbox* deals with real models so that poles are either real or come in conjugate pairs

$$\{\lambda, \bar{\lambda}\} = \{a \pm ib\} = \{-\zeta\omega \pm \omega\sqrt{1-\zeta^2}\}$$

Poles can be stored in three accepted formats which are automatically recognized by `ii_pof`(see warnings below for exceptions).



Format 1: a column vector of complex poles. `ii_pof` puts the pairs of complex conjugate poles $\lambda, \bar{\lambda}$ first and real poles at the end

$$xxx\texttt{po} = \left\{\begin{array}{c} \lambda_1 \\ \bar{\lambda}_1 \\ \vdots \\ \lambda_{Re} \\ \vdots \end{array}\right\} \text{ for example}$$

```
po=[-0.0200 + 1.9999i
    -0.0200 - 1.9999i
    -1.0000]
```

Because non-real poles come in conjugate pairs with conjugate eigenvectors, it is generally easier to only view the positive-imaginary and real poles, as done in the two other formats.

Format 2: real and imaginary part

$$xxx\texttt{po} = \left[\begin{array}{cc} a & b \\ \vdots & \vdots \end{array}\right] \text{ for example}$$

```
po=[-0.0200 1.9999
    -1.0000 0.0000]
```

Format 3: frequency ω and damping ratio ζ

$$\text{po} = \begin{bmatrix} \omega_1 & \zeta_1 \\ \vdots & \vdots \end{bmatrix} \quad \text{for example} \quad \begin{array}{l} \text{po=[2.0000 0.0100} \\ \textit{hbox} \quad -1.0000\ 1.0000] \end{array}$$

To **sort** the poles while changing format use an arbitrary third argument `SortFlag`.

Warnings The input format is recognized automatically. An **error** is however found for poles in input format 2 (real and imaginary) with all imaginary below 1 and all real parts positive (unstable poles). In this rare case you should change your frequency unit so that some of the imaginary parts are above 1.

Real poles are always put at the end. If you create your own residue matrices, make sure that there is no mismatch between the pole and residue order (the format for storing residues is described in section 2.6).

See also `idcom`, `id_rc`, `ii_plp`

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 command of `m_elastic` : `Database` and `Dbval`. Then materials supported by `m_elastic` are described.

`[Database,Dbval] [-unit TY] [,MatiD]] Name`

A material property function is expected to store a number of standard materials. See section 7.3 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');
pl=fe_mat('convert SITM',pl);
pl=m_elastic(pl,'dbval -unit TM 102 steel')
```

The default material is steel.

`Material description`

Then the commands, that a material function is expected to implement, are described to let users develop their own material handling functions.

`m_elastic` supports the following material subtypes

`1 : standard isotropic`

Standard isotropic materials 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 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 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

3-D Anisotropic solid 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.

`m_elastic`, and all material handling functions, must support the following commands

See also Section 5.1.4, section 7.3, `fe_mat`, `p_shell`

m_piezo

Purpose Material function for piezoelectric solids

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

$$\left\{\begin{array}{c}\epsilon_x\\ \epsilon_y\\ \epsilon_z\\ \gamma_{yz}\\ \gamma_{zx}\\ \gamma_{xy}\\ E_x\\ E_y\\ E_z\end{array}\right\}=\left[\begin{array}{cccc}N,x & 0 & 0 & 0\\ 0 & N,y & 0 & 0\\ 0 & 0 & N,z & 0\\ 0 & N,z & N,y & 0\\ N,z & 0 & N,x & 0\\ N,y & N,x & 0 & 0\\ 0 & 0 & 0 & N,x\\ 0 & 0 & 0 & N,y\\ 0 & 0 & 0 & N,z\end{array}\right]\left\{\begin{array}{c}u\\ v\\ w\\ \phi\end{array}\right\} \tag{9.16}$$

`p_solid` `ConstPiezo` supports integration constant building for piezo electric volumes.

nasread

Purpose Read results from outputs of the MSC/NASTRAN finite element code.

Syntax

```
out = nasread('FileName','Command')
```

Description The base `nasread` reads bulk data deck (NASTRAN input), results in the `.f06` text file (no longer supported).

The `nas2up` extension (part of the FEMLink extension of SDT) allows direct reading of model and result information in OUTPUT2 files generated using NASTRAN `PARAM,POST,-i` cards. This is the most efficient and accurate method to import NASTRAN results for post-processing (visualization with `feplot`, normal model handling with `nor2ss`, ...) or parameterized model handling with `upcom`. Available commands are

`Bulk file`

`model=nasread('FileName','bulk')` reads NASTRAN bulk files for nodes (grid points), element description matrix, material and element properties, and coordinate transformations, MPC, SPC, DMIG, SETS, ...

Use `'BulkNo'` for a file with no `BEGIN BULK` card. Unsupported cards are displayed to let you know what was not read. You can omit the `'bulk'` command when the file name has the `.dat` or `.bdf` extension.

Each row of the `bas.bas` output argument contains the description of a coordinate system.

The current element equivalence table is

NASTRAN	*SDT*
CELAS1, CELAS2, RBAR	`celas`
RBE2	`rigid`
RBE3	`rbe3` in `Case`
CONROD	`bar1`
CBAR, CBEAM, CROD	`beam1`
CBUSH	`cbush`
CSHEAR	`quad4`
CONM1, CONM2	`mass2`
CHEXA	`hexa8, hexa20`
CPENTA	`penta6, penta15`
CTETRA	`tetra4, tetra10`
CTRIA3, CTRIAR	`tria3`
CTRIA6	`tria6`
CQUAD4, CQUADR	`quad4`
CQUAD8	`quadb`

Details on properties are given under `naswrite WritePLIL`. NASTRAN Scalar points are treated as standard SDT nodes with the scalar DOF being set to DOF `.01` (this has been tested for nodes, DMIG and MPC).

`OUTPUT2 binary (FEMLink)`

`[model,out]=nasread('FileName','output2')` reads `output2` *binary output format* for tables, matrices and labels. You can omit the `output2` command if the file names end with `2`. The output `model` is a model data structure described in section 7.6. If deformations are present in the binary file, the are save in the `model.def` field (see section 7.8).

The optional `out` argument is a cell array with fields the following fields

`.name`	Header data block name (table, matrix) or label (label)
`.dname`	Data block name (table, matrix) or NASTRAN header (label)
`.data`	cell array with logical records (tables), matrix (matrix), empty (label)
`.trl`	Trailer (7 integers) followed by record 3 data if any (for table and matrix), date (for label)

The basic `nasread` only reads integer valued tables. The `nas2up` extension to the SDT provides translation for the following tables

`GEOM1`	nodes with support for local coordinates and output of nodes in global coordinates
`GEOM2`	elements with translation to SDT model description matrix (see `bulk` command).
`GEOM4`	translates constraints (`MPC`, `OMIT`, `SPC`) and rigid links (`RBAR`, `RBE1`, `RBE2`, `RBE3`, `RROD`, ...) to SDT model description matrix
`GPDT`	with use of `GPL` and `CSTM` to obtain nodes in global coordinates
`KDICT`	reading of element mass (`MDICT`, `MELM`) and stiffness (`KDICT`, `KELM`) matrix dictionaries and transformation of a type 3 superelement handled by `upcom`. This is typically obtained from NASTRAN with `PARAM,POST,-4`. To choose the file name use `Up.file='FileName';Up=nasread(Up,'Output2.op2');`
`MPT`	material property tables
`OUG`	transformation of shapes (modes, time response, static response, ...) to `.def` field.
`OES`	tables of element stresses or strains.

This translation allows direct reading/translation of output generated with NASTRAN `PARAM,POST` commands simply using `out=nasread('FileName.op2')`. For model and modeshapes, use `PARAM,POST,-1`. For model and element matrices use `PARAM,POST,-4` or `PARAM,POST,-5` (see `BuildUp` command below).

`BuildUp,BuildOrLoad (FEMLink)`

A standard use of FEMLink is to import a model including element matrices to be used later with `upcom`. You must first run NASTRAN SOL103 with `PARAM,POST,-4` to generate the appropriate `.op2` file (note that you must include the geometry in the file that is not use `PARAM,OGEOM,NO`). Assuming that you have saved the bulk file and the `.op2` result in the same directory with the same name (different extension), then

```
Up=nasread('FileName.blk','buildup')
```

reads the bulk and `.op2` file to generate a superelement saved in `FileName.mat`.

It is necessary to read the bulk because linear constraints are not saved in `op2` file during the NASTRAN run. If you have no such constraints, you can read the `.op2` only with `Up=upcom('load FileName);Up=nasread(Up,'FileName.op2')`.

The `BuildOrLoad` command is used to generate the `upcom` file on the first run and simply load it if it already exists.

```
nasread('FileName.blk','BuildOrLoad') % result in global variable Up
```

`OUTPUT4 binary`

`out=nasread('FileName','output4')` reads `output4` *binary output format* for matrices. The result `out` is a cell array containing matrix names and values stored as MATLAB sparse matrices.

All double precision matrix types are now supported. If you encounter any problem, ask for a patch which will be provided promptly.

`.f06 output (obsolete)`

ASCII reading in `.f06` files is slow and often generates round-off errors. You should thus consider reading binary OUTPUT2 files, which is now the only supported format.

`nasread('FileName','matprt')` reads matrix by (`DMAP` command `MATPRT`). Matrices printed to an `.f06` output file using the `MATPRT` command are read and saved in global variables whose name is displayed.

`nasread('F','tabpt')` reads tables printed with the `DMAP` command `TABPT`). Tables printed to a `.f06` output file using the `TABPT` command are read and saved in global variables whose name is displayed. Different records of the table are saved as matrices named `TabName`*I* (where *I* is the record number).

`[mode,mdof,freq]=nasread('FileName','realmodes')` reads normal mode analysis results (NASTRAN solution 103) from a `.f06` output file. The frequencies (in rad/s) are stored as a vector `freq`, the modes and corresponding DOF definition vector in `mode` and `mdof`. The matrices `mode`, `mdof`, and `freq` are returned as output arguments of `nasread` as shown above.

```
[vector,mdof]=nasread('filename','vectortype')
```

reads vectors defined at nodes from a `.f06` output file. Supported vectors are displacement (`displacement`), applied load vector (`oload`) and grid point stress (`gpstress`).

See also `naswrite`, `ufread`, `nas2up`, importing models

naswrite

Purpose Formatted ASCII output to MSC/NASTRAN bulk data deck. Most commands are only supported with FEMLink.

Syntax

```
naswrite('FileName',node,elt,pl,il)
naswrite('FileName','command', ...)
naswrite(fid,'command', ...)
```

Description `naswrite` appends its output to the file `FileName` or creates it, if it does not exist. You can also provide a handle `fid` to a file that you opened with `fopen`. `fid=1` can be used to have a screen output.

`EditBulk`

Supports bulk file editing. Calls take the form `nas2up('EditBulk',InFile,edits,Outfile)`, where `InFile` and `OutFile` are file names and `edits` is a cell array with four columns giving command, begin tag, end tag, and data. Accepted commands are

`Before`	inserts data before the `BeginTag`.
`Remove`	removes a given card. Warning this does not yet handle multiple line cards.
`Set`	used to set parameter and assign values **Syntax** For example

```
edits={'Set','PARAM','POST','-2'};
rootname='my_job';
f0={'OUTPUT4',sprintf('%s_mkekvr.op4',rootname),'NEW',40,'DELETE
    'OUTPUT4',sprintf('%s_TR.op4',rootname),'NEW',41,'DELETE'};
edits(end+1,1:4)={'set','ASSIGN','',f0}
```

`model`

`naswrite('FileName',model)` the nominal call, it writes everything possible : nodes, elements, material properties, case information (boundary conditions, loads, etc.). For example `naswrite(1,femesh('testquad4'))`.

The following information present in model stack is supported

- curves as TABLED1 cards if some curves are declared in the model.Stack see fe_curve for the format).
- Fixed DOFs as SPC1 cards if the model case contains FixDof and/or KeepDof entries. FixDof,AutoSPC is ignored if it exists.
- Multiple point constraints as MPC cards if the model case contains MPC entries.
- coordinate systems as CORDi cards if model.bas is defined (see basis for the format).

The obsolete call naswrite('FileName',node,elt,pl,il) is still supported.

node,elt

You can also write nodes and elements using the low level calls but this will not allow fixes in material/element property numbers or writting of case information.

```
femesh('testquad4')
fid=1 % fid=fopen('FileName');
naswrite(fid,'node',FEnode)
naswrite(fid,'node',FEnode)
%fclose(fid)
```

dmig

DMIG writting is supported through calls of the form naswrite(fid,'dmigwrite NAME',mat,mdof). For example

```
 naswrite(1,'dmigwrite KAAT',rand(3),[1:3]'+.01)
```

A 'nastran','dmig' entry in model.Stack, where the data is a cell array where each row gives name, DOF and matrix, will also be written. You can then add these matrices to your model by adding cards of the form K2GG=KAAT to you NASTRAN case.

job

NASTRAN job handling on a remote server from the Matlab command line is partially supported. You are expected to have ssh and scp installed on your computer. On windows, it is assumed that you have access to these commands using CYGWIN. You first need to define your preferences

```
setpref('FEMLink','CopyFcn','scp');
setpref('FEMLink','RunNastran','NASTRAN');
setpref('FEMLink','RemoteShell','ssh');
setpref('FEMLink','RemoteDir','/tmp2/NASTRAN');
setpref('FEMLink','RemoteUserHost','user@myhost.com')
setpref('FEMLink','DmapDir',fullfile(fileparts(which('nasread')),'dmap'))
```

You can then run a job using `nas2up('joball','BulkFileName.dat')`. Other commands are `jobcpto` that copies files to the remote directory and `jobcprom` which fetches files.

You can define a job handler customized to your needs and still use the `nas2up` calls for portability by defining `setpref('FEMLink','NASTRANJobHandler','FunctionName')`

`Wop4`

Matrix writing to `OUTPUT4` format. You provide a cell array with one matrix per row, names in first column and matrix in second column. The optional byte swapping argument can be used to write matrices for use on a computer with another binary format.

```
kv=speye(20);
ByteSwap=0;  % No Byte Swapping needed
nas2up('File.op4',{'kv',kv},ByteSwap);
```

For `ByteSwap` you can also specify `ieee-le` for little endian (Intel PC) or `ieee-be` depending on the architecture NASTRAN will be running on.

`WriteFreqLoad`

`edits=naswrite('Target.bdf','WriteFreqLoad',model)` (or the equivalent `nas2up` call when the file is already open as show below) writes loads defined in `model` (and generated with `Load=fe_load(model)`) as a series of cards. `FREQ1` for load frequencies, `TABLED1` for the associated curve, `RLOAD1` to define the loaded DOFs and `DAREA` for the spatial information about the load. The return `edits` argument is the entry that can be used to insert the associated subcase information in a nominal bulk.

The identifiers for the loads are supposed to be defined as `Case.Stack{j1,end}.ID` fields.

```
% Generate a model with sets of point loads
model=demosdt('Demo ubeam dofload noplot')
% Define the desired frequencies for output
```

```
model=stack_set(model,'info','Freq', ...
   struct('ID',101,'data',linspace(0,10,12)));
fid=1 % fid=fopen('FileName');
edits=nas2up('writefreqload',fid,model);
fprintf('%s\n',edits{end}{:}); % Main bulk to be modified with EditBulk
%fclose(fid)
```

Write[Curve,Set,SetC,Uset]

WriteCurve lets you easily generate NASTRAN curve tables.

WriteSet lets you easily generate NASTRAN node and elements sets associated with node and element selection commands. WriteSetC formats the sets for use in the case control section rather than the bulk.

WriteUset generates DOFs sets.

```
model=demosdt('demogartfe');
fid=1; % display on screen (otherwise use FOPEN to open file)
nas2up('WriteSet',fid,3000,model,'findnode x>.8');
selections={'zone_1','group 1';'zone_2','group 2:3'};
nas2up('WriteSet',fid,2000,model,selections);

curves={'curve','Sine',fe_curve('testsin -id1',linspace(0,pi,10)),
        'curve','Exp.',fe_curve('testexp -id100',linspace(0,1,30))};
nas2up('WriteCurve',fid,curves)
DOF=feutil('getdof',model);
nas2up('WriteUset U4',fid,DOF(1:20))
```

WritePLIL

The WritePLIL is used to resolve identifier issues in MatId and ProId (elements in SDT have both a MatId and an ProID while in NASTRAN they only have a ProId with the element property information pointing to material entries). While this command is typically used indirectly while writing a full model, you may want to access it directly. For example

```
 model=demosdt('demogartfe');
 nas2up('Writeplil',1,model);
```

The implementation of p_solid properties is somewhat different in NASTRAN and SDT, thus for a il row giving

`[ProID type Coordm In Stress Isop Fctn  ]`

In a NASTRAN Bulk file, `In` is either a string or an integer. If it is an integer, this property is the same in `il`. If it is a string equal to resp. `TWO` or `THREE`, this property is equal to resp. 2 or 3 in `il`.

In a NASTRAN Bulk file, `Stress` is either a string or an integer. If it is an integer, this property is the same in `il`. If it is a string equal `GAUSS`, this property is equal to 1 in `il`.

In a NASTRAN Bulk file, `Isop` is either a string or an integer. If it is an integer, this property is the same in `il`. If it is a string equal `FULL`, this property is equal to 1 in `il`.

If `Fctn` is equal to `FLUID` in the NASTRAN Bulk file, it is equal to 1 in `il` and elements are read as `flui*` elements.

See also `nasread`, `ufread`, `ufwrite`

nor2res, nor2ss, nor2xf

Purpose Transformations from normal mode models to other model formats.

Syntax

```
[res,po,psib,cpsi] = nor2res( ... )
               RES = nor2res( ... )
         [a,b,c,d] = nor2ss ( ... )
               SYS = nor2ss ( ... )
        [xf,IDopt] = nor2xf ( ... )
             XF(i) = nor2xf ( ... ,'struct')
               ... = nor2.. (ga,om,pb,cp, ... )
               ... = nor2.. (DEF,ga,CASE, ... )
               ... = nor2ss ( ... , ind,fc,type)
               ... = nor2xf ( ... , w,ind,fc,type)
```

Description Normal mode models are second order models detailed in the theory section below. `nor2res`, `nor2ss`, and `nor2xf` provide a number of transformations from the normal mode form to residue, state-space, and transfer function formats.

The normal mode model is specified using either high level structure arguments `DEF,ga,Case` or low level numeric arguments `om,ga,pb,cp`. Additional arguments `w,ind,fc,type` can or must be specified depending on the desired output.

`DEF,ga,CASE`

The normal mode shapes are can be given in a `DEF` a structure with fields `.def`, `.DOF`, `.data` (see section 7.8).

These mode shapes are assumed mass normalized and the first column of the `.data` field is assumed to give modal frequencies **in Hz**. They can be computed with `fe_eig` or imported from an external FEM code (see section 5.5.1).

Damping (argument `ga`) can be declared in different ways

- modal damping ratio can be given in `DEF.data(:,2)`
- a vector `damp` of modal damping ratio can be given as the second argument
- a modal damping matrix `ga` can be given as the second argument. Note that this modal damping matrix is assumed to use frequency units consistent with

the specified frequencies. Thus a physical viscous damping matrix will need to be divided by 2*pi (see demo_fe).

- hysteretic modal damping is not systematically supported since it leads to complex valued state-space models. You can compute FRFs with an hysteretic modal damping model using

  ```
  def.data=sqrt(real(def.data.^2)).*sqrt(1+i*damp*2);
  IIxh=nor2xf(def,[],Case,w,'hz');
  ```

  as illustrated in section 2.3.2.

Inputs and outputs are described by a CASE (see section 5.2) or a model containing a Case (see section 5.1). Giving the model is needed when inputs correspond to distributed loads (FVol or FSurf case entries detailed under fe_load). SensDof are the only output entries currently supported (see fe_case).

Note that DofSet entries are handled as acceleration inputs. The basis described by DEF must allow a correct representation of these inputs. This can be achieved by using a basis containing static corrections for unit displacements or loads on the interface (see fe2ss CraigBampton or Free commands). A proper basis can also be generated using acceleration inputs at single nodes where a large seismic mass is added to the model. This solution is easier to implement when dealing with external FEM codes.

Here is a sample call using this format

```
load sdt_gart;
Case=fe_case('DofLoad','Force',[4.03;55.03;2.03], ...
             'SensDof','Sensors',[4 55 30]'+.03);
DEF=struct('def',md0,'DOF',mdof,'data',f0);
IIw=linspace(5,70,500)';
IIxf = nor2xf(DEF,.01,Case,IIw,'hz acc');
IIpo=f0;iiplot
```

When using distributed loads (pressure, etc.), the model elements are needed to define the load so that the model rather than a Case must be given as in the following example

```
model = femesh('testubeam plot');
def=fe_eig(model,[106 20 10000 11 1e-5]);

%Pressure load
```

```
data=struct('sel','x==-.5', ...
    'eltsel','withnode {z>1.25}','def',1,'DOF',.19);
model=fe_case(model,'addtocase','Fsurf','Surface load',data)
%Sensors
model=fe_case(model,'addtocase','sensdof','Sensors',[50:54]'+.03);

fe_case(model,'info')

IIw=linspace(10,240,460)';
IIxf=nor2xf(def,0.05,model,IIw,'hz');
iiplot;
```

om,ga,pb,cp

Standard low level arguments om (modal stiffness matrix), ga (modal viscous damping matrix), pb (modal controlability) and cp (modal observability) used to describe *normal mode models* are detailed in section section 2.2.

A typical call using this format

```
load sdt_gart
b = fe_c(mdof,[4.03;55.03])'; c = fe_c(mdof,[1 30 40]'+.03);
IIw=linspace(5,70,500)';
IIxf = nor2xf(f0*2*pi,0.01,md0'*b,c*md0,IIw*2*pi);
IIpo=f0;iiplot
```

w,ind,fc,type

Other arguments are

w	frequencies (**in rad/s** unless Hz is specified in type) where the FRF should be computed (for nor2xf)
ind	(optional) gives the indices of modes to be retained
fc	(optional) roll-off frequency or correction mode poles for static correction modes (for load input only)
type	(optional) is a string that can contain. 'Hz' to specify that w and wj are given in Hz. Non diagonal om or ga are always given in rad/s. 'dis', 'vel', or 'acc' are used to obtain displacement (default), velocity or acceleration output. 'struct' is used to obtain a structure compatible with database wrappers (see xfopt).

res

nor2res returns a complex mode model in the residue form

$$[\alpha(s)] = \sum_{j=1}^{2N} \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j} = \sum_{j=1}^{2N} \frac{[R_j]}{s-\lambda_j}$$

This routine is particularly useful to recreate results in the identified residue form res for comparison with direct identification results from id_rc.

Pole residue models are always assumed to correspond to force to displacement transfer functions. Acceleration input or velocity, acceleration output specifications are thus ignored.

ss

nor2ss returns state-space models (see the theory section below).

When no roll-off frequency is specified, nor2ss introduces a correction, **for displacement only**, in the state-space models through the use of a non-zero d term. If a roll-off frequency fc is given, the static correction is introduced in the state-space model through the use of additional high frequency modes. Unlike the non-zero D term which it replaces, this correction also allows to correct for velocity contributions of truncated modes.

You can also specify fc as a series of poles (as many as inputs) given in the frequency/damping format (see ii_pof).

xf

nor2xf computes FRF (from u to y) associated to the normal mode model. When used with modal frequencies freq and a subset of the modes (specified by a non empty ind), nor2xf introduces static corrections for the truncated modes.

Theory

The basic normal mode form associated with load inputs $[b]\{u\}$ is (see section 2.2)

$$\begin{gathered}\left[[I]\,s^2 + [\Gamma]\,s + [\Omega^2]\right]_{NP\times NP}\{(s)\} = \left[\phi^T b\right]_{NP\times NA}\{u(s)\}_{NA\times 1}\\ \{y(s)\} = [c\phi]_{NS\times NP}\{p(s)\}_{NP\times 1}\end{gathered}$$

where the coordinates p are such that the mass is the identity matrix and the stiffness is the diagonal matrix of frequencies squared.

The associated state-space model has the form

$$\left\{ \begin{array}{c} \dot{p}\,(t) \\ \ddot{p}\,(t) \end{array} \right\} = \left[\begin{array}{cc} [0] & [I] \\ -\left[\backslash \Omega^2 \backslash\right] & -[\Gamma] \end{array} \right] \left\{ \begin{array}{c} p\,(t) \\ \dot{p}\,(t) \end{array} \right\} + \left[\begin{array}{c} 0 \\ \phi^T b \end{array} \right] \{u(t)\}$$

$$\{y\} = [c\phi \;\; 0] \left\{ \begin{array}{c} p\,(t) \\ \dot{p}\,(t) \end{array} \right\} + [0]\,\{u(t)\}$$

When used with modal frequencies wj and a subset of the modes (specified by ind), nor2ss introduces static corrections for the truncated modes. When requesting velocity or acceleration output, static correction can only be included by using additional modes.

In cases with displacement output only, the static corrections are ranked by decreasing contribution (using a SVD of the d term). You can thus look at the input shape matrix b to see whether all corrections are needed.

nor2ss (and nor2xf by calling nor2ss) supports the creation of state-space models of transmissibilities (transfer functions from acceleration input to displacement, velocity or acceleration. For such models, one builds a transformation such that the inputs u_a associated with imposed accelerations correspond to states

$$\left\{ \begin{array}{c} u_a \\ q_c \end{array} \right\} = [T_I \;\; T_C]\,\{p\}$$

and solves the fixed interface eigenvalue problem

$$\left[T_C^T \Omega T_C - \omega_{jC}^2 T_C^T I T_C\right] \{\phi_{jC}\} = \{0\}$$

leading to basis $\left[T_I \;\; \hat{T}_C\right] = [T_I \;\; T_C\,[\phi_{jC}]]$ which is used to build the state space model

$$\left\{ \begin{array}{c} \dot{u} \\ \dot{q}_C \\ \ddot{u} \\ \ddot{q}_C \end{array} \right\} = \left[\begin{array}{cc} [0] & [I] \\ \left[\begin{array}{c} 0 \\ -\hat{T}_C^T \Omega \left[T_I \;\; \hat{T}_C\right] \end{array} \right] & \left[\begin{array}{c} 0 \\ -\hat{T}_C^T \Gamma \left[T_I \;\; \hat{T}_C\right] \end{array} \right] \end{array} \right] \left\{ \begin{array}{c} u \\ q_C \\ \dot{u} \\ \dot{q}_C \end{array} \right\} +$$

$$\left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 0 & I \\ \hat{T}_C^T b & \hat{T}_C^T T_I \end{array} \right] \left\{ \begin{array}{c} u_F \\ \ddot{u}_a \end{array} \right\}$$

$$\{y\} = \left[cT_I \;\; c\hat{T}_C \;\; 0 \;\; 0\right] \left\{ \begin{array}{c} u_a \\ q_C \\ \dot{u}_a \\ \dot{q}_C \end{array} \right\} + [0] \left\{ \begin{array}{c} u_F \\ \ddot{u}_a \end{array} \right\}$$

nor2res, nor2ss, nor2xf

Simple adjustments lead to velocity and acceleration outputs.

When using acceleration input, care must be taken that the initial shapes of the normal mode model form an appropriate basis. This can be achieved by using a basis containing static corrections for unit displacements or loads on the interface (see `fe2ss CraigBampton` or `Free` commands) or a seismic mass technique.

See also `res2nor`, `id_nor`, `fe_c`, `psi2nor`

`demo_fe`

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')`.

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.

spfmex

`spfmex` is a sparse multi-frontal solver based on `spooles` a compiled version is provided with SDT distributions.

sp_util

The skyline matrix storage is a traditional form to store the sparse symmetric matrices corresponding to FE models. For a full symmetric matrix `kfull`

```
kfull=[1  2
          10  5  8  14
              6  0  1
                 9  7
        sym.        11  19
                        20]
```

The non-zero elements of each column that are above the diagonal are stored sequentially in the data field `k.data` from the diagonal up (this is known as the reverse Jenning's representation) and the indices of the elements of `k` corresponding to diagonal elements of the full matrix are stored in an index field `k.ind`. Here

```
k.data = [1; 10; 2; 6; 5; 9; 0; 8; 11; 7; 1; 14; 20; 19; 0]
k.ind  = [1; 2; 4; 6; 9; 13; 15];
```

For easier manipulations and as shown above, it is assumed in the that the index field `k.ind` has one more element than the number of columns of `kfull` whose value is the index of a zero which is added at the end of the data field `k.data`.

If you have imported the `ind` and `data` fields from an external code, `ks = ofact (data, ind)` will create the ofact object. You can then go back to the MATLAB sparse format using `sparse(ks)` (but this needs to be done before the matrix is factored when solving a static problem).

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`, `fe_reduc`

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*.

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 7.4). 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 5.1.4. For a programmers reference on formats used to describe element properties see section 7.4.

`1 :  standard`

`p_beam` currently only supports a single format (`fe_mat` property subtype)

```
[ProID   type   J I1 I2 A   k1 k2 Lump]
```

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)
`k2`	(optional) shear factor for direction 2
`lump`	(optional) request for lumped mass model

`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. [33] 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 [53]. No correction for rotational inertia of sections is used.

See also Section 5.1.4, section 7.4, `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');
```

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.

For example, you will obtain the element property row with EltId 100 associated with a .1 thick Kirchhoff shell or the corresponding Mindlin plate use

```
il = p_shell('database 100 MindLin .1')
il = p_shell('dbval 100 kirchhoff .1')
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 7.4). 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 5.1.4. For a programmers reference on formats used to describe element properties see section 7.4.

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 correspond 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.
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 (use p_shell('Co to display.

$$\left\{\begin{array}{c}\epsilon_{xx}\\ \epsilon_{yy}\\ 2\epsilon_{xy}\\ \kappa_{xx}\\ \kappa_{yy}\\ 2\kappa_{xy}\\ \gamma_{xz}\\ \gamma_{yz}\end{array}\right\} = \left[\begin{array}{ccccc} N,x & 0 & 0 & 0 & 0\\ 0 & N,y & 0 & 0 & 0\\ N,y & N,x & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & -N,x\\ 0 & 0 & 0 & N,y & 0\\ 0 & 0 & 0 & N,x & -N,y\\ 0 & 0 & N,x & 0 & N\\ 0 & 0 & N,y & -N & 0\end{array}\right] \left\{\begin{array}{c}u\\ v\\ w\\ ru\\ rw\end{array}\right\} \tag{9.17}$$

```
2 :  composite
   [ProID type   Z0 NSM SB FT TREF GE LAM MatId1 T1 Theta1 SOUT1 ...]
```

`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.

$$\left\{\begin{array}{c} N \\ M \\ Q \end{array}\right\} = \left[\begin{array}{ccc} A & B & 0 \\ B & D & 0 \\ 0 & 0 & F \end{array}\right] \left\{\begin{array}{c} \epsilon \\ \kappa \\ \gamma \end{array}\right\} \tag{9.18}$$

See also Section 5.1.4, section 7.4, `fe_mat`

p_solid

Purpose Element property function for solid 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 7.4). Element property functions such as `p_solid` support graphical editing of properties and a database of standard properties.

Accepted value in database are `Full 2x2x2` and `Reduced shear`.

For a tutorial on material/element property handling see section 5.1.4. For a programmers reference on formats used to describe element properties see section 7.4.

Examples of database property construction

```
il=p_solid([100 fe_mat('p_solid','SI',1) 0 3 0 2], ...
           'dbval 101 Full 2x2x2');
il=fe_mat('convert SITM',il);
il=p_solid(il,'dbval -unit TM 2 Reduced shear')
```

`Subtype 1 :  3D volume element`

`[ProID type Coordm In Stress Isop Fctn  ]`

`ProID`	Property identification number
`type`	Identifier obtained with `fe_mat('p_solid,'SI',1)`
`Coordm`	Identification number of the material coordinates system
`In`	Integration rule selection. See the result of `integrule(ElemF,In)`. Integration rule selection is only supported by the `*b.m` element families. 0 selects the default for the element. -2 is integration at nodes.
`Stress`	Location selection for stress output (NOT USED)
`Isop`	Integration scheme (will be used to select shear protection mechanisms)
`Fctn`	Fluid element flag (this is a NASTRAN flag and will never be used in OpenFEM)

`p_solid ConstSolid` supports integration constant building for general elastic volumes with strain defined by (see `hexa8b constants`)

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \end{Bmatrix} \begin{bmatrix} N,x & 0 & 0 \\ 0 & N,y & 0 \\ 0 & 0 & N,z \\ 0 & N,z & N,y \\ N,z & 0 & N,x \\ N,y & N,x & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{9.19}$$

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 volume elements inherited from MODULEF order shear stresses differently $\sigma_{xy}, \sigma_{yz}, \sigma_{zx}$, in `fe_stres` this is accounted for by the definition of the proper `TensorTopology` matrix.

`p_solid ConstFluid` supports integration constant building for acoustic volumes with strain defined by (see `flui4 constants`)

$$\begin{Bmatrix} p,x \\ p,y \\ p,z \end{Bmatrix} = \begin{bmatrix} N,x \\ N,y \\ N,z \end{bmatrix} \begin{Bmatrix} p \end{Bmatrix} \tag{9.20}$$

`Subtype 2 :  2D volume element`

`[ProId Type Form N In]`

p_solid

`ProID`	Property identification number
`type`	Identifier obtained with `fe_mat('p_solid,'SI',2)`
`Form`	Formulation (0 plane strain, 1 plane stress, 2 axisymetric)
`N`	Fourier harmonic for axisymetric elements that support it
`In`	Integration rule selection. See the result of `integrule(ElemF,In)`. Integration rule selection is only supported by the `*b.m` element families. 0 selects the default for the element. -2 is integration at nodes.

See also Section 5.1.4, section 7.4, `fe_mat`

p_spring

Purpose Element property function for spring and rigid elements

Syntax

```
il=p_spring('default')
il=p_spring('database MatId Value')
pl=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 7.4).

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 -unit TM 2 xxx')
```

`p_spring` currently only supports one subtype

`1 : standard`

`[ProID type  k c b m]`

`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`

NOT DOCUMENTED YET

See also Section 5.1.4, section 7.4, `fe_mat`

perm2sdt

Purpose Read results from outputs of the PERMAS (V7.0) finite element code.

Syntax

```
out = perm2sdt('Read Model_FileName')
out = perm2sdt('Read Result_FileName')
out = perm2sdt('merge',model)
out = perm2sdt('binary.mtl Matrix_FileName')
out = perm2sdt('ascii.mtl Matrix_FileName')
```

Description The `perm2sdt`function reads PERMAS model, result and matrices files. Binary and ASCII files are supported.

`Model files`

To read a FE model, use the following syntax: `model = perm2sdt('Read FileName')`

To deal with sub-components, you may use the `merge` command.

The current element equivalence table is

SDT	*PERMAS*
`mass2`	MASS3, MASS6, X1GEN6
`bar1`	FLA2
`beam1`	PLOTL2, BECOC, BECOS, BECOP, BETOP, BETAC, FD-PIPE2, X2GEN6
`celas`	SPRING3, SPRING6, SPRING1, X2STIFF3
`t3p`	TRIM3
`tria3`	TRIA3, TRIA3K, TRIA3S, FSINTA3
`quad4`	QUAD4, FSINTA4, QUAD4S, PLOTA4, SHELL4
`flui4`	FLTET4
`tetra4`	TET4
`tetra10`	TET10
`penta6`	PENTA6, FLPENT6
`hexa8`	HEXE8, FLHEX8
`pyra5`	PYRA5, FLPYR5

`Merging model`

The `merge` command integrates subcomponents into the main model.

`Result files`

The syntax is

`perm2sdt('read result_file')`

`Matrix files`

`perm2sdt`reads binary and ASCII `.mtl` file format. The syntax is

`perm2sdt('binary.mtl File.mtl')` for binary files and and

`perm2sdt('ascii.mtl File.mtl')` for ASCII files.

See also `naswrite`, `ufread`, importing models

psi2nor

Purpose Estimation of normal modes from a set of scaled complex modes.

Syntax

```
[wj,ga,cps,pbs] = psi2nor(po,cp)
[wj,ga,cps,pbs] = psi2nor(po,cp,ncol,NoCommentFlag)
```

Description `psi2nor` should generally be used through `id_nor`. For cases with as many and more sensors than modes, `psi2nor` gives, as proposed in Ref. [5], a proper approximation of the complex mode outputs `cp`$= [c]\,[\psi]$ (obtained using `id_rm`), and uses the then exact transformation from complex to normal modes to define the normal mode properties (modal frequencies `wj`, non-proportional damping matrix `ga`, input `pbs`$= [\phi]^T\,[b]$ and output `cps`$= [c]\,[\phi]$ matrices).

The argument `ncol` allows the user to specify the numbers of a restricted set of outputs taken to have a collocated input (`pbs=cps(ncol,:)'`).

If used with less than four arguments (not using the `NoCommentFlag` input argument), `psi2nor` will display two indications of the amount of approximation introduced by using the proper complex modes. For the complex mode matrix ψ_T (of dimensions NT by $2NT$ because of complex conjugate modes), the properness condition is given by $\psi_T\psi_T^T = 0$. In general, identified modes do not verify this condition and the norm $\|\psi_T\psi_T^T\|$ is displayed

```
The norm of psi*psi' is 3.416e-03 instead of 0
```

and for well identified modes this norm should be small (10^{-3} for example). The algorithm in `psi2nor` computes a modification $\Delta\psi$ so that $\tilde{\psi}_T = \psi_T + \Delta\psi$ verifies the properness condition $\tilde{\psi}_T\tilde{\psi}_T^T = 0$. The mean and maximal values of `abs(dpsi./psi)` are displayed as an indication of how large a modification was introduced

```
The changes implied by the use of proper cplx modes are
0.502 maximum and 0.122 on average
```

The modified modes do not necessarily correspond to a positive-definite mass matrix. If such is not the case, the modal damping matrix cannot be determined and this results in an error. Quite often, a non-positive-definite mass matrix corresponds to a scaling error in the complex modeshapes and one should verify that the identification process (identification of the complex mode residues with `id_rc` and determination of scaled complex mode outputs with `id_rm`) has been accurately done.

Warnings The complex modal input is assumed to be properly scaled with reciprocity constraints (see `id_rm`). After the transformation the normal mode input/output matrices verify the same reciprocity constraints. This guarantees in particular that they correspond to mass-normalized analytical normal modes.

For lightly damped structures, the average phase of this complex modal output should be close to the -45^o line (a warning is given if this is not true). In particular a sign change between collocated inputs and outputs leads to complex modal outputs on the $+45^o$ line.

Collocated force to displacement transfer functions have phase between 0 and -180^o, if this is not verified in the data, one cannot expect the scaling of `id_rm` to be appropriate and should not use `psi2nor`.

See also `id_rm`, `id_nor`, `id_rc`, `res2nor`, `nor2xf`, `nor2ss`, the `demo_id` demonstration

qbode

Purpose Frequency response functions (in the `xf` format) for linear systems.

Syntax

```
xf = qbode(a,b,c,d,w)
xf = qbode(ss,w)
xf = qbode(num,den,w)
XF = qbode( ... ,'struct')
     qbode( ... ,'plot')
```

Description For state-space models described by matrices `a`, `b`, `c`, `d`, or the LTI state-space object `sys` (see *Control System Toolbox*), `qbode` uses an eigenvalue decomposition of `a` to compute, in a minimum amount of time, all the FRF `xf` at the frequency points `w`

$$\texttt{xf} = [C]\left(s\left[\backslash I_{\backslash}\right] - [A]\right)^{-1}[B] + [D]$$

The result is stored in the `xf` format (see details page 40). .

`qbode` **will not work if your model is not diagonalizable**. A specific algorithm was developed to deal with systems with rigid-body modes (double pole at zero associated to non-diagonalizable matrices). This algorithm will not, however, indicate the presence of incoherent `b` and `c` matrices. In other cases, you should then use the direct routines `res2xf`, `nor2xf`, etc. or the `bode` function of the *Control System Toolbox*.

For the polynomial models `num`, `den` (see details page 39), `qbode` computes the FRF at the frequency points `w`

$$\texttt{xf} = \frac{\texttt{num}(j\omega)}{\texttt{den}(j\omega)}$$

Warnings

- All the SISO FRF of the system are computed simultaneously and the complex values of the FRF returned. This approach is good for speed but not always well numerically conditioned when using state space models not generated by the *SDT*.
- As for all functions that do not have access to options (`IDopt` for identification and `Up.copt` for FE model update) frequencies are assumed to be given in the mathematical default (rad/s). If your frequencies `w` are given in Hz, use `qbode(sys,w*2*pi)`.

- Numerical conditioning problems may appear for systems with several poles at zero.

See also `demo_fe`, `res2xf`, `nor2xf`, and `bode` of the *Control System Toolbox*

res2nor

Purpose Approximate transformation from complex residues to normal mode residue or proportionally damped normal mode forms.

Syntax

```
[Rres,po,Ridopt] = res2nor(Cres,po,Cidopt)
[wj,ga,cp,pb]  = res2nor(Cres,po,Cidopt)
```

Description The contributions of a pair of conjugate complex modes (complex conjugate poles λ and residues R) can be combined as follows

$$\frac{[R]}{s-\lambda}+\frac{[\bar{R}]}{s-\bar{\lambda}}=2\frac{(s\mathrm{Re}(R))+(\zeta\omega\mathrm{Re}(R)-\omega\sqrt{1-\zeta^2}\mathrm{Im}(R))}{s^2+2\zeta\omega s+\omega^2}$$

Under the assumption of proportional damping, the term $s\mathrm{Re}(R)$ should be zero. `res2nor`, assuming that this is approximately true, sets to zero the contribution in s and outputs the normal mode residues `Rres` and the options `Ridopt` with `Ridopt.Fit = 'Normal'`.

When the four arguments of a normal mode model (see `nor` page 28) are used as output arguments, the function `id_rm` is used to extract the input `pbs` and output `cps` shape matrices from the normal mode residues while the frequencies `wj` and damping matrix `ga` are deduced from the poles.

Warning This function assumes that a proportionally damped model will allow an accurate representation of the response. For more accurate results use the function `id_nor` or identify using real residues (`id_rc` with `IDopt.Fit='Normal'`).

See also `id_rm`, `id_rc`, `id_nor`, `res2ss`, `res2xf`

res2ss, ss2res

Purpose Transformations between the residue `res` and state-space `ss` forms.

Syntax

```
SYS            = res2ss(RES)
SYS            = res2ss(RES,'AllIO')
[a,b,c,d]      = res2ss(res,po,IDopt)
RES            = ss2res(SYS)
[res,po,IDopt] = ss2res(a,b,c,d)
```

Description The functions `res2ss` and `ss2res` provide transformations between the complex / normal mode residue forms `res` (see section 2.6) and the state space forms (see section 2.4). You can use either high level calls with data structures or low level calls providing each argument

```
demosdt('demo gartid est')
SYS = res2ss(XF(5));
RES = ss2res(SYS);
[a,b,c,d] = res2ss(XF(5).res,XF(5).po,XF(5).idopt);
```

Important properties and limitations of these transformations are

`res2ss`

- The residue model should be minimal (a problem for MIMO systems). The function `id_rm` is used within `res2ss` to obtain a minimal model (see section 3.4.1). To obtain models with multiple poles use `id_rm` to generate `new_res` and `new_po` matrices.
- `IDopt.Reciprocity='1 FRF'` or `MIMO id_rm` then also constrains the system to be reciprocal, this may lead to differences between the residue and state-space models.
- The constructed state-space model corresponds to a displacement output.
- Low frequency corrections are incorporated in the state-space model by adding a number (minimum of `ns` and `na`) of poles at 0.

  Asymptotic corrections (see `IDopt.ResidualTerms`) other than the constant and s^{-2} are not included.
- See below for the expression of the transformation.
- The `'AllIo'` input can be used to return all input/output pairs when assuming reciprocity.

ss2res

- Contributions of rigid-body modes are put as a correction (so that the pole at zero does not appear). A real pole at 0 is not added to account for contributions in *1/s*.
- To the exception of contributions of rigid body modes, the state-space model must be diagonalizable (a property verified by state-space representations of structural systems).

Theory

For control design or simulation based on identification results, the minimal model resulting from id_rm is usually sufficient (there is no need to refer to the normal modes). The state-space form is then the reference model form.

As shown in section 3.4.1, the residue matrix can be decomposed into a dyad formed of a column vector (the modal output), and a row vector (the modal input). From these two matrices, one derives the $[B]$ and $[C]$ matrices of a real parameter state-space description of the system with a bloc diagonal $[A]$ matrix

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} = \begin{bmatrix} [0] & \left[\backslash I_\backslash\right] \\ -\left[\backslash \omega_{j\backslash}^2\right] & -\left[\backslash 2\zeta_j\omega_{j\backslash}\right] \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} + \begin{Bmatrix} B_1 \\ B_2 \end{Bmatrix} \{u(t)\}$$
$$\{y(t)\} = [C_1 \;\; C_2] \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix}$$

where the blocks of matrices B_1, B_2, C_1, C_2 are given by

$$\begin{Bmatrix} C_{1j} \\ C_{2j} \end{Bmatrix} = \frac{1}{\omega_j\sqrt{1-\zeta_j^2}} \begin{bmatrix} \omega_j\sqrt{1-\zeta_j^2} & \zeta_j\omega_j \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{Re}\,(c\psi_j) \\ \mathrm{Im}\,(c\psi_j) \end{bmatrix}$$
$$\begin{Bmatrix} B_{j1} \\ B_{j2} \end{Bmatrix} = 2\left[\mathrm{Re}\left(\psi_j^T b\right) \;\; \mathrm{Im}\left(\psi_j^T b\right)\right] \begin{bmatrix} 1 & 0 \\ \zeta_j\omega_j & -\omega_j\sqrt{1-\zeta_j^2} \end{bmatrix}$$

Form the state space model thus obtained, FRFs in the xf format can be readily obtained using qbode. If the state space model is not needed, it is faster to use res2xf to generate these FRFs.

See also

demo_fe, res2xf, res2nor, qbode, id_rm, id_rc

res2tf, res2xf

Purpose

Create the polynomial representation associated to a residue model.
Compute the FRF corresponding to a residue model.

Syntax

```
[num,den] = res2tf(res,po,IDopt)
xf        = res2xf(res,po,w,IDopt)
xf        = res2xf(res,po,w,IDopt,RetInd)
```

Description

For a set of residues `res` and poles `po` (see `res` page 37) , `res2tf` generates the corresponding polynomial transfer function representation (see `tf` page 39)).

For a set of residues `res` and poles `po`, `res2xf` generates the corresponding FRFs evaluated at the frequency points `w`. `res2xf` uses the options `IDopt.Residual`, `.DataType`, `AbscissaUnits`, `PoleUnits`, `FittingModel`. (see `idopt` for details).

The FRF generated correspond to the FRF used for identification with `id_rc` except for the complex residue model with positive imaginary poles only `IDopt.Fit='Posit'` where the contributions of the complex conjugate poles are added.

For MIMO systems, `res2tf` and `res2xf` do not restrict the pole multiplicity. These functions and the `res2ss`, `qbode` sequence are thus not perfectly equivalent. A unit multiplicity residue model for which the two approaches are equivalent can be obtained using the matrices `new_res` and `new_po` generated by `id_rm`

```
[psib,cpsi,new_res,new_po]=id_rm(IIres,IIpo,IDopt,[1 1 1 1]);
IIxh = res2xf(new_res,new_po,IIw,IDopt);
```

The use of `id_rm` is demonstrated in `demo_id`.

See also

`res2ss`, `res2nor`, `qbode`, `id_rm`, `id_rc`

rms

Purpose Computes the RMS response of the given frequency response function `xf` or auto-spectra `a` to a unity white noise input over the frequency range `w`.

Syntax

```
rm = rms(t,w)
rm = rms(a,w,1)
```

Description The presence of a third input argument indicates that an auto-spectrum `a` is used (instead of frequency response function `xf`).

A trapezoidal integration is used to estimate the root mean squared response

$$\texttt{rms} = \sqrt{\frac{1}{2\pi}\int_{\omega_1}^{\omega_2}|t(\omega)|^2 d\omega} = \sqrt{\frac{1}{2\pi}\int_{\omega_1}^{\omega_2} a(\omega) d\omega}$$

If `xf` is a matrix containing several column FRF, the output is a row with the RMS response for each column.

Warning If only positive frequencies are used in `w`, the results are multiplied by 2 to account for negative frequencies.

See also `ii_cost`

setlines

Purpose Line color and style sequencing utility.

Syntax

```
setlines
setlines(ColorMap,LineSequence)
setlines(ColorMapName,LineSequence,MarkerSequence)
```

Description The M-by-3 `ColorMap` or `ColorMapName` (standard color maps such as `jet`, `hsv`, etc.) is used as color order in place or the `ColorMap` given in the `ColorOrder` axis property (which is used as a default).

The optional `LineSequence` is a matrix giving the `linestyle` ordering whose default is `['- ';'--';'-.';':  ']`.

The optional `MarkerSequence` is a matrix giving the `marker` ordering. Its default is empty (marker property is not set).

For all the axes in the current figure, `setlines` finds solid lines and modifies the `Color`, `LineStyle` and `Marker` properties according the arguments given or the defaults. Special care is taken to remain compatible with plots generated by `feplot` and `iiplot`.

`setlines` is typically used to modify line styles before printing. Examples would be

```
setlines k
setlines([],'-','ox+*s')
setlines(get(gca,'colororder'),':','o+^>')
```

sdplot

Purpose Curve plotting interface.

Syntax

```
sdplot
```

This interface is the future replacement of `iiplot`. It does not currently have all the functionality.

`Axes[Scale,Lim]`

These commands are used to specify default axes properties for the current `sdplot` axis. Thus `AxesScale xLin` or `AxesScale zLog`, `AxesLim xauto`, `AxesLim xManual` are acceptable commands.

`AxSub`

`AxSub` commands are used to determine axes that are displayed in the `sdplot` figure.

sdplot(uf,'axsub ','XF(1)');

`db[Append,DefaultXf]`

`db` commands give access to the curve database in `uf.dbase`. `uf.dbase.Stack` is a three column cell array where each row gives `'curve',CurveName,Data`.

`sdplot('DbDefaultXF')` appends pointers to the global variable `XF` used by `iiplot`. Thus allowing a backward compatibility mode with `iiplot`.

`GfCurrent`

`uf=sdplot('_gfcurrent')` returns the contents of the current `sdplot` figure. `uf` is a data structure with fields

`Open`

`Show ...`

Show commands are used to initialize basic plots and possibly toggle what is shown in the current axis. The typical calls take the form

```
sdplot('ShowFrf',{'DatasetName1','DatasetName2'})
```

Basic plots are

`frf`	Basic frequency response function with response versus frequency, x and y labels.

The following show commands toggle what is displayed in the current axis

`abs`	absolute value
`imag`	imaginary part
`phase`	wrapped phase
`phaseu`	unwrapped phase
`real`	real part (used to display real responses such as time traces)
`rvsi`	real versus imaginary (used for Nyquist plots in particular)

See also `iiplot`

sdtdef

Purpose Internal function used to handle default definitions.

Syntax

```
sdtdef('info')
sdtdef('ConstantName',Value)
sdtdef('ConstantName')
```

Description For an exact list of current defaults use `sdtdef('info')`. To reset values to factory defaults use `sdtdef('factory')`.

Values that you are likely to need changing are

`avi`	cell array of default AVI properties, see the MATLAB `avifile` command.
`DefaultFeplot`	cell array of default `feplot` figure properties. For MATLAB versions earlier than 6.5, the OpenGL driver is buggy so you will typically want to set the value with `sdtdef('DefaultFeplot',{'Renderer' 'zbuffer' ... 'doublebuffer' 'on'})`
`epsl`	tolerance on node coincidence used by `femesh`, `feutil`. Defaults to 1e-6 which is generally OK except for MEMS applications, ...

The following MATLAB preferences can also be used to customize SDT behavior for your particular needs

SDT	DefaultZeta	Default value for the viscous damping ratio. The nominal valu 1e-2.
SDT	KikeMemSize	Memory in megabytes used to switch to an out-of-core savin element matrix dictionaries.
SDT	tempdir	can be used to specify a directory different than the tempdi turned by MATLAB. This is typically used to specify a faster l disk.
SDT	OutOfCoreBufferSize	Memory in bytes used to decide switching to an out-of-core cedure. This is currently used by nasread when reading l OUTPUT2 files.
FEMLink	TextUnix	set to 1 if text needs to be converted to UNIX (rather than D mode before any transfer to another machine.
FEMLink	NASTRAN	NASTRAN version. This is used to implement version depen writing of NASTRAN files.

sdth

Purpose Class constructor for *SDT* handle objects.

Description The *Structural Dynamics Toolbox* now supports *SDT handles* (`sdth` objects). Currently implemented types for `sdth` objects are

`SDTRoot`	global context information used by the toolbox
`IDopt`	identification options (see `idopt`)
`FeplotFig`	`feplot` figure handle
`IiplotFig`	`iiplot` figure handle
`VectCor`	Vector correlation handle (see `ii_mac`)
`XF`	Database wrapper (see `xfopt`)

SDT `handles` are wrapper objects used to give easier access to user interface functions. Thus `IDopt` displays a detailed information of current identification options rather than the numeric values really used.

Only advanced programmers should really need access to the internal structure of *SDT* `handles`. The fixed fields of the object are `opt`, `type`, `data`, `GHandle` (if the `sdth` object is stored in a graphical object), and `vfields`.

Most of the information is stored in the variable field storage field `vfields`.

See also `feplot`, `idopt`, `iiplot`, `ii_mac`, `xfopt`

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.

sp_util

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(Up,typ);
[StackRows,index]=stack_get(Up,typ,name);
Up=stack_set(Up,typ,name,val)
Up=stack_rm(Up,typ,name);
Up=stack_rm(Up,typ);
Up=stack_rm(Up,'',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 7.6 or section 7.7 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.

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
[Val,ind]=stack_get(Case,'DofSet')
% Access value
Case.Stack{ind(1),3}
Val{1,3}
```

ufread

Purpose Read from Universal Files.

Syntax

```
ufread
UFS = ufread('FileName')
UFS = ufread('FileList*.uff')
```

Description The Universal File Format is a set of ASCII file formats widely used to exchange analysis and test data. As detailed below `ufread` supports test related UFF (15 grid point, 55 analysis data at node, 58 response data at DOF) and with the FEMLink extension FEM related datasets.

`ufread` with no arguments opens a GUI to let you select a file and displays the result using `feplot` and/or `iiplot`. `UFS = ufread('FileName')` returns either a FEM model (if only model information is given) or a database wrapper `UFS` pointing to the universal files present in `FileName` grouped by blocks of files read as a single dataset in the *SDT* (all FRFs of a given test, all trace lines of a given structure, etc.). You can specify a file list using the `*` character in the file name.

You get a summary of the database contents by displaying `UFS`

```
>> UFS

UFS = UFF Database Wrapper for file 'example.uff'

{1} [.Node (local) 107x7, .Elt (local)  7x156] : model
 2  [.w    (UFF)   512x1, .xf  (UFF)    512x3] : response data
 3  [.po   (local)  11x2, .res (local) 11x318] : shape data
```

which indicates the content of each dataset in the database wrapper, the current data set between braces `{ }`, the type and size of the main data fields. For response data (UFF type 58), the data is only imported when you refer to it (`UFS(i)` call) but it is imported every time you do so unless you force loading into memory using `UFS(i)=UFS(i)`.

The `UFS` object gives you direct access to the data in each field. In the example above, you can display the modeshapes using

```
cf       = feplot;
cf.model = UFS(1);
cf.def   = UFS(3);
```

When loading response data, you may want to transfer all options from the universal file to the standard database wrapper `XF` using calls of the form `XF(2)=UFS(3)`.

15 Grid point

Grid points stored in a node matrix (see node page 142) in a `UFS(i).Node` field.

The format is a (4I10,1P3E13.5) record for each node with fields
`[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 (this option is not currently used), `GID` is a node group number (zero or any positive integer), and `x y z` are the coordinates.

55 Analysis data at node

Analysis data at nodes are characterized by poles `.po` and residues `.res` (corresponding to DOFs `.dof`) and correspond to shape at DOF datasets in *SDT* database wrappers (see more info under the `xfopt` help).

The information below gives a short description of the universal file format. You are encouraged to look at comments in the `ufread` and `ufwrite` source codes if you want more details.

Header1	(80A1). The UFF header lines are stored in the .header field
Header2	(80A1)
Header3	(80A1) DD-MMM-YY and HH:MM:SS with format (9A1,1X,8A1)
Header4	(80A1)
Header5	(80A1)
Fun	(6I10) This is stored in the database wrapper .fun field Model (0 Unknown, 1 Structural, 2 Heat Transfer, 3 Fluid Flow) Analysis (0 Unknown, 1 Static, 2 Normal Mode, 3 Complex eigenvalue first order, 4 Transient, 5 Frequency Response, 6 Buckling, 7 Complex eigenvalue second order Field 0 Unknown, 1 Scalar, 2: Tx Ty Tz, 3: Tx Ty Tz Rx Ry Rz, 4: Sxx Sxy Syy Sxz Syz Szz, 5: Sxx Syx Szx Sxy Syy Szy Sxz Syz Szz FieldType see list with xfopt('_fieldtype') Format 2 Real, 5 Complex NDV Number Of Data Values Per Node
SpeInt	(8I10) NumberOfIntegers on this line (3-N are type specific), NumberOfReals on the next line, SpeInt type specific integers (see table below for details)
SpeRea	Type specific real parameters
NodeID	(I10) Node number
Data	(6E13.5) Data At This Node : NDV Real Or Complex Values (real imaginary for data 1, ...) Records 9 And 10 Are Repeated For Each Node.

Type specific values depend on the Signification value and are stored in the .r55 field of the database wrapper.

0 Unknown	`[ 1 1 ID Number]`
	`[0.0]`
1 Static	`[1 1 LoadCase]`
	`[0.0]`
2 Normal model	`[2 4 LoadCase ModeNumber]`
	`[FreqHz ModalMass DampRatioViscous DampRatioHysteretic]`
3 Complex eigenvalue	`[2 6 LoadCase ModeNumber]`
	`[ReLambda ImLambda ReModalA ImModalA ReModalB ImModalB]`
4 Transient	`[2 1 LoadCase TimeStep]`
	`[TimeSeconds]`
5 Frequency response	`[2 1 LoadCase FreqStepNumber]`
	`[FrequencyHz]`
6 Buckling	`[1 1 LoadCase]`
	`[Eigenvalue]`

58 Function at nodal DOF

Functions at nodal DOF are characterized by frequencies `w`, a data set `xf`, as well as other options. The information below gives a short description of the universal file format. You are encouraged to look at comments in the `ufread` and `ufwrite` source codes if you want more details. Functions at nodal DOFs are grouped by type and stored in response data sets of `UFS`.

Information about how the UFF data is stored in *SDT* database wrappers can be found in the `xfopt` help.

`Header1`	(80A1) Function description
`Header2`	(80A1) Run Identification
`Header3`	(80A1) Time stamp DD-MMM-YY and HH:MM:SS with format (9A1,1X,8A1)
`Header4`	(80A1) Load Case Name
`Header5`	(80A1)
`DOFID`	2(I5,I10),2(1X,10A1,I10,I4) with values `FunType` (list with `xfopt('_funtype')`), `FunID`, `VerID`, `LoadCase` (0 single point), `ResponseGroup` (`NONE` if unused, `ResponseNodeID`, `ResponseDofID 1:6` correspond to *SDT* DOFs `.01` to `.06`, `-1:-6` to *SDT* DOFs `.07` to `.12` `ReferenceGroup`, `ReferenceNodeID`, `ReferenceDofID 1:6`. These are only relevant if `LoadCase is zero`

DataForm	(3I10,3E13.5)
	DFormat (2 : real, single precision, 4 : real, double precision, 5 : complex, single precision, 6 : complex, double precision), NumberOfDataPoints, XSpacing (0 - uneven, 1 - even (no abscissa values stored)), XMinimum (0.0 if uneven), XStep (0.0 if spacing uneven), ZAxisValue (0.0 if unused)
XDataForm	(I10,3I5,2(1X,20A1)) DataType (list with xfopt('_datatype')), lue length unit exponents, fue force, tue temperature, AxisLabel, AxisUnits Note : exponents are used to define dimensions. Thus Energy (Force * Length) has [fue lue tue]=[1 1 0]. This information is generally redundant with DataType.
YNDataForm	Ordinate (or ordinate numerator) Data Form (same as XDataForm
YDDataForm	Ordinate Denominator Data Characteristics
ZDataForm	Z-axis Data Characteristics
DataValue	a series of x value (if uneven x spacing, always with format E13.5), real part, imaginary part (if exists) with precision (E13.5 or E20.12) depending on DFormat.

82, Trace Line

Trace Line matrix LDraw where each non-empty row corresponds to a line to be traced. All trace lines, are stored as element groups of UFS(1).Elt.

LDraw can be used to create animated deformation plots using feplot.

Opt	(3I10) LineNumber, NumberOfNodes, Color
Label	(80A1) Identification for the line
Header3	(8I10) node numbers with 0 for discontinuities

(,1:2)	[*NumberOfNodes GroupID*]
(,3:82)	[*LineName*] (which should correspond to the group name)
(,83:end)	[*NodeNumbers*] (*NumberOfNodes* of them, with zeros to break the line)

151, Header

Header stored as a string matrix header (with 7 rows).

780, 2412, Elements

These universal file formats are supported by the SDT FEMLink extension.

SDT	UNV element (UNV Id)
beam1	rod (11), linear beam (21)
tria3	thin shell lin triangle (91), plane stress lin tri (41), plan strain lin tri (51), flat plate lin triangle (74)
tria6	thin shell para tri (92), plane stress para tri (42), plane strain para tri (51), flat plate para tri (62), membrane para tri (72)
quad4	thin shell lin quad (94), plane stress lin quad (44), plane strain lin quad (54), flat plate lin quad (64), membrane lin quad (71)
quadb	thin shell para quad (95), plane stress para quad (54), plane strain para quad(55), flat plate para quad (65), membrane para quad(75)
tetra4	solid lin tetra (111)
tetra10	solid para tetra (118)
penta6	solid lin wedge (112)
penta15	solid para wedge (113)
hexa8	solid lin brick (115)
hexa20	solid para brick (116)
rigid	rigid element (122)
bar1	node-node trans spring (136), node-node rot spring (137)
mass2	lumped mass (161)

773, 1710 Material Database

These universal file formats are supported by the SDT FEMLink extension.

All materials properties are read, but obviously only those currently supported by the SDT are translated to the corresponding row format (see `m_elastic` and section 7.4).

772, 788, 789, 2437, Element Properties

These universal file formats are supported by the SDT FEMLink extension.

All element (physical) properties are read, but obviously only those currently supported by the SDT are translated to the corresponding row format (see `p_beam`, `p_shell`, section 7.3).

`2414, Analysis data`

These universal file formats are supported by the SDT FEMLink extension.

Note that the list of FEMLink supported dataset is likely to change between manual editions. Please get in touch with SDTools if a dataset you want to read is not supported.

See also `nasread`, `ufwrite`, `xfopt`

ufwrite

Purpose Write to a Universal File.

Syntax

```
ufwrite(FileName,UFS,i)
ufwrite(FileName,model)
```

Description You can export to UFF using the `feplot` and `iiplot` export menus.
`ufwrite(FileName,UFS,`*i*`)` appends the dataset *i* from a database wrapper `UFS` to the file `FileName`. Database wrappers are described in the `xfopt` reference section.
`ufwrite(FileName,model)` can be used to export FEM models.

For datasets representing

- models, `ufwrite` writes a UFF of type 15 for the nodes and a trace line (UFF 82) for test wire frames (all `EGID` negative) or without FEMLink. With FEMLink, nodes are written in UFF 2411 format and elements in UFF 2412.
- response data, `ufwrite` writes a *response at DOF* (UFF 58) for each column of the response set.
- shape data, `ufwrite` writes a *data at nodal DOF* (UFF 55) for each row in the shape data set.

Starting from scratch, you define an empty database wrapper `DB=xfopt('empty')`. You can then copy data sets from the standard database wrapper `XF` (previously initialized by `iiplot` or `xfopt`) using `DB(`*i*`)=XF(`*j*`)`. You can also build a new data set by giving its fields (see `xfopt` for the fields for the three supported dataset types). The following would be a typical example

```
UF=xfopt('empty')
UF(1)={'node',FEnode,'elt',FEelt};
UF(2)={'w',IIw,'xf',IIxf};
UF(3)={'po',IIres,'res',IIres,'dof',XFdof};
```

Once the database wrapper built, `ufwrite('NewFile',UF,1:3)` will write the three datasets.

With `iiplot`, you can use the standard database wrapper `XF` to change properties as needed then write selected datasets to a file. For example,

```
load gartid
```

```
iiplot
XF(1).x='frequency';  % modify data set properties
XF(1).yn='accele';
iicom('sub');         % reinitialize plot to check
st=fullfile(getpref('SDT','tempdir'),'test.uf');
ufwrite(st,XF,1);
XF(7)={'node',FEnode,'elt',FEelt};
ufwrite(st,XF,7);
UFS=ufread(FileName); % reread the UFF to check result
```

Note that you can edit these properties graphically in the `iiplot properties ...` figure.

See also `ufread`, `iiplot`, `nasread`

upcom

Purpose

User interface function for parameterized superelements.

Description

The `upcom` interface supports type 3 superelements which handle parameterization by storing element matrix dictionaries and thus allowing reassembly of mass and stiffness matrices computed as weighted sums of element matrices (6.26).

By default, `upcom` uses a special purpose superelement stored in the **global variable** `Up`. You can however use more than one type 3 superelement by providing the appropriate variables as input/output arguments. `upcom('info')` applies to `Up` whereas `upcom(model,'info')` applies to `model`.

The `par` commands are used to dynamically relate the element matrix weights to physical parameters thus allowing fairly complex parametric studies on families of models. The main objective for `upcom` is to enable finite element model updating, but it can also be used for optimization and all problems using with families of models or hysteretic damping modeling as illustrated in section 2.3.2.

The following paragraphs detail calling formats for commands supported by `upcom` and are followed by an explanation of the signification of the fields of `Up` (see the `commode` help for hints on how to build commands and understand the variants discussed in this help).

More details on how these commands are typically sequenced are given in the *Tutorial* section 6.3 and section 6.4.

The implementation of the `upcom` interface has undergone major revisions for *SDT 5.0* so that it is not fully backward compatible. In particular the handling of parameters and the `assemble` calls have changed.

Commands

`Clear, Load` *`File`* `, Save` *`File`*

`upcom('clear')` clears the global variable `Up`. `upcom('load` *`File`*`')` loads the superelement fields from *`File`*`.mat` and creates the file if it does not currently exist. `upcom('save` *`File`*`')` makes sure that the current values of the various fields are saved in *`File`*`.mat`. Certain commands automatically save the superelement but efficiency mandates not to do it all the time. The working directory field `Up.wd` lets you work in a directory that differs from the directory where the file is actually located.

`Assemble [,m,k] [,coef` *`cur`*`],[,delta` *`i`*`][,NoT][,Point]`

`[m,k] = upcom('assemble')` returns the mass and stiffness parameters associated with the parameters by the last `parcoef` command.

`Assemble Coef` *`cur`* uses the parameter values `cur` for the assembly. `Assemble CoefNone` does not use any parameter definitions (all the element matrices are used with a unit weighting coefficient). `AssembleMind` uses columns 5 and 6 of `Up.mind` for element matrix coefficients.

`Assemble Delta` *`i`* assembles the derivative of matrices with respect to parameter *i*. To assemple a derivative with non zero components on more than one parameter, use `[dm,dk]=upcom('assemble delta',dirp)` where `dirp` (with $Npar$ rows) characterizes the amplitude of the derivative on each parameter for the current change. `dirp` can for example be used to describe simultaneous changes in mass and stiffness parameters.

`k=upcom('assemble k coef 2 3')` only assembles the stiffness with parameter coefficients set to 2 and 3. Similarly, `dm=upcom('assemble m delta 2')` will assemble the mass derivative with respect to parameter 2.

The `NoT` modifier can be used to prevent the default projection of the matrices on the master DOFs defined by the current case.

The `Point` modifier can be used return the `v_handle` object pointing to the non assembled matrix. This matrix can then be used in `feutilb('tkt')` and `feutilb('a*b')` out of core operations.

`ComputeMode [ ,full,reduced] [,`*`eig_opt`*`]`

`[mode,freq] = upcom('ComputeMode')` assembles the model mass and stiffness based on current model parameters (see the `parcoef` command) and computes modes. The optional `full` or `reduced` can be used to change the current default (see the `opt` command). The optional *`eig_opt`* can be used to call `fe_eig` with options other than the current defaults (see the `opt` command).

```
upcom('load GartUp');
def = upcom('computemode full 105 10 1e3');
```

For reduced model computations, the outputs are `[moder,freq,modefull]`.

`ComputeModal [ ,full,reduced]`

`[IIxe,mode,freq]=upcom('ComputeModal',damp,b,c,IIw)` computes the normal modes and static corrections for inputs `b` of the full or reduced order models based on the full or reduced model. `nor2xf` is then used to compute the frequency response associated with the input shape matrix `b` (using full order model DOFs), the output shape matrix `c` and the frequency points given in `IIw` (units specified in `UP.copt(1,3)`, set with the `OptUnit` command).

ComputeFRF

`[IIxe]=upcom('ComputeFrf',b,c,IIw)` computes FRFs associated the input shape matrix `b` (using full order model DOFs), the output shape matrix `c` and the frequency points given in `IIw` (units specified in `Up.copt(1,3)`). It does not compute modes and is thus faster than `ComputeModal` for a full order model and a few frequency points.

Ener [m, k]

`ener = upcom('ener k',def)` computes the strain energy in each element for the deformations `def`. `ener` is a data structure with fields `.IndInElt` specifying the element associated with each energy row described in the `.data` field. You can display the kinetic energy in an arbitrary element selection of a structure, using a call of the form

```
cf.sel={'group6','colordata elt',upcom('ener m','group6',mode)};
```

Fix

`upcom('fix0')` eliminates DOFs with no stiffness contribution. `upcom('fix',adof)` only retains DOFs selected by `adof`.

This command is rather inefficient and you should eliminate DOFs with `FixDOF` case entries (see `fe_case`) or assemble directly with the desired DOFs (specify `adof` in the `SetNominal` command).

Get

Information about the superelement is stored in fields of the global variable `Up`. The easiest way to access those fields is to make the variable local to your workspace (use `global Up`) and to access the fields directly. The superelement also has pseudo-fields `mi,me,ki,ke` which are always stored in `Up.file`. Commands of the form `load(Up.file,'ke')` are used to get them.

femesh

upcom femesh copies Up.Elt to FEelt and Up.Node to FEnode so that femesh commands can be applied to the model.

IndInElt

upcom('IndInElt') returns a vector giving the row position in Up.Elt of each row in Up.mind. This is in particular used for color coded energy plots which should now take the form

```
feplot('ColorDataElt',upcom('eners',res),upcom('indinelt'));
```

Although it is typically easier to use high level calls of the form

```
upcom('plotelt'); cf=feplot;model=femesh('test 2bay');
cf.sel={'groupall','colordata enerk'};
```

Info [,par,elt]

upcom('info') prints information about the current content of Up: size of full and reduced model, values of parameters currently declared, types, etc.

InfoPar details currently defined parameters. InfoElt details the model.

Opt

upcom('opt *Name Value* ') sets the option *Name* to a given *Value*. Thus upcom ('opt gPrint 11') sets the general printout level to 11 (maximum). Accepted names and values are detailed in the Up.copt field description below.

Par Coef

The value of each physical parameter declared using ParStack commands is described by a row of coefficients following the format

```
[type cur min max vtype]
```

Accepted parameter types are the following

1	stiffness proportional to parameter value. This is the case for a variable Young's modulus.
2	mass proportional to parameter. This is the case for a variable mass density.
3	variable thickness. Currently only valid for `quad4` and `quadb` elements. `tria3` elements can be handled with degenerate `quad4`. Element groups with variable thickness must be declared at assembly (`SetNominal` command).

The following columns are `cur`rent, `min`, `max` and `nom`inal values. `vtype` deals with the type of variation 1 (linear), 2 (log not implemented yet).

`upcom('parcoef',cur)` is used to set current values (`cur` must be a vector of length the number of declared parameters), while `upcom('parcoef',par)` also sets min, max and vtype values. You can also use `[cur,par]=upcom('parcoef')` or `par=upcom('parcoefpar')` to obtain current values or the parameter value matrix.

A parameter initialization would thus be as follows (but type specification with `ParStackAdd` commands is the preferred strategy)

```
upcom('load GartUp');
upcom('ParStackreset')
upcom('ParStackadd k','Tail','group3');
upcom('ParStackadd t','Constrained Layer','group6');
par = [ 1   1.0 0.1 3.0   1
        3   1.0 0.1 3.0   1 ];
upcom('parcoef',par);
upcom('info par');
[cur,par]=upcom('parcoef')
```

Note that to prevent user errors, `upcom` does not allow parameter overlap for the same type of matrix (modification of the modulus and/or the thickness of the same element by two distinct parameters).

ParRed

`upcom('par red',T)` projects the current full order model with the currently declared parameters on the basis `T`. Typical reduction bases are discussed in section 6.1.7 and an example is shown in the `gartup` demo. Matrices to be projected are selected based on the currently declared variable parameters in such a way that projected reduced model is able to make predictions for new values of the parameters.

ParStack [add *type values*,reset]

These commands allow the creation of a parameter definition stack. Each parameter is given a type (k for stiffness, m for mass, t for thickness) optional current, min and max values, a name, and an element selection command.

```
upcom('load GartUp');
upcom('ParStackreset')
upcom('ParStackadd k 1.0 0.5 2.0','Tail','group3');
upcom('ParStackadd t 1.0 0.9 1.1','Constrained Layer','group6');
upcom('parcoef',[1.2 1.3]);
upcom('info par');
```

upcom('ParStackreset') reinitializes the parameter stack. In the example, the current parameter values are modified using the ParCoef command.

Parameters are stored in the current case stack which you can select with

```
[Case,name]=fe_case(Up,'getcase');
des=stack_get(Case,'par');
```

des is a cell array where each row has the form {'par','name',data} with data containing fields

.sel	string or cell array allowing selection of elements affected by the parameter
.coef	vector of parameter coefficients (see format description under the upcom ParCoef command).
.pdir	Boolean vector giving the positions of affected elements in Up.mind

ParTable

tt=upcom('partable') returns a cell array of string describing the parameters currently declared. This cell array is useful to generate formatted outputs for inclusion in various reports using comstr(tt,-17,'excel') for example.

PlotElt

upcom plotelt initializes a feplot figure displaying the model in upcom. If Up has deformations defined in a .def field, these are shown using cf=feplot;cf.def=Up.

Profile [,fix]

Renumbers DOFs and pseudo-fields mi,me,ki,ke using symrcm to minimize matrix bandwidth. ProfileFix eliminates DOFs with no stiffness on the diagonal at the same time. `upcom('ProfileFix',fdof)` profiles and eliminates DOFs in `fdof` and DOFs with no stiffness on the diagonal.

Support for case entries (see `fe_case`) makes this command obsolete.

`SensMode [,reduced]`

`[fsen,mdsen,mode,freq] = upcom('SensMode',dirp,indm,T)` returns frequency and modeshape sensitivities of modes with indices given in `indm` for modifications described by `dirp`.

For a model with NP parameters (declared with the `ParStack` commands), `dirp` is a matrix with $Npar$ rows where each column describe a case of parameter changes of the form `par = dirp(:,j)`. The default for `dirp` the identity matrix (unit change in the direction of each parameter).

The optional argument `T` can be used to give an estimate of modeshapes at the current design point. If `T` is given the modes are not computed which saves time but decreases accuracy if the modes are not exact.

`fsen` gives, for modes `indm`, the sensitivities of modal frequencies squared to all parameters (one column of `fSen` per parameter). `mdsen` stores the modeshape sensitivities sequentially (sensitivities of modes in `indm` to parameter 1, parameter 2, ...).

When modeshape sensitivities are not desired (output is `[fsen]` or `[fsen, mode, freq]`), they are not computed which takes much less computational time.

By default `SensMode` uses the full order model. The first order correction to the modal method discussed in Ref. [38] is used. You can access the reduced order model sensitivities using `SensModeReduced` but should be aware that accuracy will then strongly depend on the basis you used for model reduction (`ParRed` command).

`SetNominal [ , t` *groups*`]`

`upcom('setnominal',model)` assembles the element mass and stiffness in `Up.file` for later reassembly using the `Assemble` command. Case information (boundary conditons, ... see `fe_case`) in `model` is saved in `Up.Stack` and will be used in assembly unless the `NoT` modifier is included in the `Assemble` command.

If the parameter that will be declared using the `ParStack` commands include thickness variations of some plate/shell elements, the model will use element sub-matrices. You thus need to declare which element groups need to have a separation in element submatrices (doing this separation takes time and requires more final storage memory so that it is not performed automatically). This declaration is done with a command of the form `SetNominal T` *`groups`* which gives a list of the groups that need separation.

Obsolete calling formats `upcom('setnominal',FEnode,FEelt,pl,il)` and `upcom('setnominal`
(where the empty argument `[]` is used for coherence with calls to `fe_mk`) are still supported but you should switch to using FEM model structures.

Fields of `Up` `Up` is a generic superelement (see description under `fe_super`) with additional fields described below. The `Up.Opt(1,4)` value specifies whether the element matrices are symmetric or not.

`Up.copt`

The *computational options* field contains the following information

```
(1,1:7) = [oMethod gPrint Units Wmin Wmax Model Step]
```

`oMethod`	optimization algorithm used for FE updates `1:` `fmins` of MATLAB (default) `2:` `fminu` of the *Optimization Toolbox* `3:` `up_min`
`gPrint`	printout level (`0` none to `11` maximum)
`Units`	for the frequency/time data vector `w` and the poles `01:` `w` in Hertz `02:` `w` in rad/s `03:` `w` time seconds `10:` `po` in Hertz `20:` `po` in rad/s example: `Up.copt(1,3) = 12` gives `w` in rad/sec and `po` in Hz
`Wmin`	index of the first frequency to be used for update
`Wmax`	index of the last frequency to be used for update
`Model`	flag for model selection (`0` full `Up`, `1` reduced `UpR`)
`Step`	step size for optimization algorithms (`foptions(18)`)

```
(2,1:5) = [eMethod nm Shift ePrint Thres MaxIte]
```

are options used for full order eigenvalue computations (see `fe_eig` for details).

```
(3,1)     = [exMethod ]
```

`exMethod` expansion method (0: static, 1: dynamic, 2: reduced basis dynamic, 3: modal, 4: reduced basis minimum residual)

`Up.mind, Up.file, Up.wd, mi, me, ki, ke`

`Up` stores element submatrices in pseudo-fields `mi,me,ki,ke` which are loaded from `Up.file` when needed and cleared immediately afterwards to optimize memory usage. The working directory `Up.wd` field is used to keep tract of the file location even if the user changes the current directory. The `upcom save` command saves all `Up` fields and pseudo-fields in the file which allows restarts using `upcom load`.

`Up.mind` is a *NElt* ×*6* matrix. The first two columns give element (sub-)matrix start and end indices for the mass matrix (positions in `mi` and `me`). Columns `3:4` give element (sub-)matrix start and end indices for the stiffness matrix (positions in `ki` and `ke`). Column 5 (6) give the coefficient associated to each element mass (stiffness) matrix. If columns `5:6` do not exist the coefficients are assumed equal to 1. The objective of these vectors is to optimize model reassembly with scalar weights on element matrices.

`Up.Node, Up.Elt, Up.pl, Up.il, Up.DOF, Up.Stack`

Model nodes (see section 7.1), elements (see section 7.2), material (see section 7.3) and element (see section 7.4) property matrices, full order model DOFs. These values are set during the assembly with the `setnominal` command.

`Up.Stack` contains additional information. In particular parameter information (see `upcom par` commands) are stored in a case (see section 7.7) saved in this field.

`Up.sens`

Sensor configuration array built using `fe_sens`. This is used for automatic test / analysis correlation during finite element update phases.

See also `fesuper`, `up_freq`, `up_ixf`

up_freq, up_ifreq

Purpose Sensitivity and iterative updates based on a comparison of modal frequencies.

Syntax

```
[coef,mode,freq]=up_freq('Method',fID,modeID,sens);
[coef,mode,freq]=up_ifreq('Method',fID,modeID,sens);
```

Description `up_freq` and `up_ifreq` seek the values `coef` of the currently declared `Up` parameters (see the `upcom ParStack` command) such that the difference between the measured `fID` and model normal mode frequencies are minimized.

Currently `'basic'` is the only *`Method`* implemented. It uses the maximum MAC (see `ii_mac`) to match test and analysis modes. To allow the MAC comparison modeshapes. You are expected to provide test modeshapes `modeID` and a sensor configuration matrix (initialized with `fe_sens`).

The cost used in both functions is given by

```
norm(new_freq(fDes(:,1))-fDes(:,2))/ norm(fDes(:,2))
```

`up_freq` uses frequency sensitivities to determine large steps. As many iterations as alternate matrices are performed. This acknowledges that the problem is really non-linear and also allows a treatment of cases with active constraints on the coefficients (minimum and maximum values for the coefficients are given in the `upcom ParStack` command).

`up_ifreq` uses any available optimization algorithm (see `upcom opt`) to minimize the cost. The approach is much slower (in particular it should always be used with a reduced model). Depending on the algorithm, the optimum found may or may not be within the constraints set in the range given in the `upcom ParStack` command.

These algorithms are very simple and should be taken as examples rather than truly working solutions. Better solutions are currently only provided through consulting services (ask for details at `info@sdtools.com`).

See also `up_ixf`, `up_ifreq`, `fe_mk`, `upcom`

up_ixf

Purpose Iterative FE model update based on the comparison of measured and predicted FRFs.

Syntax

```
[jump]=up_ixf('basic',b,c,IIw,IIxf,indw)
```

Description `up_ixf` seeks the values `coef` of the currently declared `Up` parameters (see the `upcom ParStack` command) such that the difference Log least-squares difference (4.5) between the desired and actual FRF is minimized. Input arguments are

`method`	Currently `'basic'` is the only *Method* implemented.
`range`	a matrix with three columns where each row gives the minimum, maximum and initial values associated the corresponding alternate matrix coefficient
`b,c`	input and output shape matrices characterizing the FRF given using the full order model DOFs. See section 2.1.
`IIw`	selected frequency points given using units characterized by `Up.copt(1,3)`
`IIxf`	reference transfer function at frequency points `IIw`
`indw`	indices of frequency points where the comparison is made. If empty all points are retained.

Currently `'basic'` is the only *Method* implemented. It uses the maximum MAC (see `ii_mac`) to match test and analysis modes. To allow the MAC comparison modeshapes. You are expected to provide test modeshapes `modeID` and a sensor configuration matrix (initialized with `fe_sens`).

`up_ixf` uses any available optimization algorithm (see `upcom opt`) to minimize the cost. Depending on the algorithm, the optimum found may or may not be within the constraints set in the range given in the `upcom ParStack` command.

This algorithm is very simple and should be taken as an example rather than an truly working solution. Better solutions are currently only provided through consulting services (ask for details at `info@sdtools.com`).

See also `up_freq`, `upcom`, `fe_mk`

v_handle

Purpose Class constructor for variable handle objects.

Description The *Structural Dynamics Toolbox* now supports variable handle objects. Which act as pointers to variables that are actually stored as global variables, user data of graphical objects, or in files.

`v_handle` objects are used to

- allow context dependent reference to a single MATLAB variable
- provide a graphic callback when modifying the object in a function or the command line.

`v_handle` objects essentially behave like global variables with the notable exception that a `clear` command only deletes the handle and not the pointed data.

Only advanced programmers should really need access to the internal structure of `v_handle`.

See also *SDT* `handle`

xfopt

Purpose User interface for database wrapper objects.

Syntax

```
xfopt command
XF(i).FieldName=FieldValue
XF(i).command='value'
XF.check
XF.save='FileName'
```

Description Database wrapper are based on *SDT* `handles` and used to access multiple sets of data. Database wrapper both contain information and point to other global variables. Thus, a database wrapper with

```
XF (global variable) = Database Wrapper (SDT Handle object)
{1} [.w  (IIw)   500x1, .xf  (IIxf)   500x10] : response (general)
```

indicates that `XF(1).w` is actually the global variable `IIw`.

`iicom`, `idcom`, and `iiplot` use the standard global variable `XF` as a common database wrapper (see `iiplot`). `ufread` and `ufwrite` also use database wrappers.

A database wrapper stores, or points to, response data in the `xf` format (see section 2.8), modes in the pole residue format (see section 2.6), or finite element models (see section 7.1 and section 7.2). The information stored for each of these data sets is detailed below.

Check, Info, Save

`XF.check` verifies the consistency of information contained in all data sets and makes corrections when needed. This is used to fill in information that may have been left blank by the user.

`disp(XF)` gives general information about the datasets. `XF(`*i*`).info` gives detailed and formatted information about the dataset in `XF(`*i*`)`. `XF(`*i*`)` only returns the actual dataset contents.

`XF.save='FileName'` saves the database wrapper as well as global variables pointed at.

XF(i).FieldName=FieldValue

xfopt supports field overload for database wrappers. This means that consistency checks are performed before actually setting a field. This is illustrated by the following example

```
iiplot
XF(1)
XF(1).x='time'; XF(1).x
XF(1).w=[1:10]';
IIw
```

XF(1) is a response dataset (with abscissa in field .w, responses in field .xf, ...).

XF(1).x='time' sets the XF(1).x field which contains a structure describing its type. Notice how you only needed to give the 'time' argument to fill in all the information. The list of supported axis types is given using xfopt('_datatype')

XF(1).w=[1:10]' sets the XF(1).w field. But since, XF(1).w contains the string 'IIw', the data is actually stored in the global variable IIw.

_FunType, _DataType, _FieldType

xfopt _FunType returns the current list of function types (given in the format specification for Universal File 58).
label=xfopt('_FunType',type) and type=xfopt('_FunType','label') are two other accepted calls.

xfopt _DataType returns the current list of data types (given in the format specification for Universal File 58). xfopt('_DataType',type) and xfopt('_DataType','label') are two other accepted calls.

For example XF.x.label='Frequency' or XF.x=18.

Data types are used to characterize axes (abscissa (x), ordinate numerator (yn), ordinate denominator (yd) and z-axis data (z)). They are characterized by the fields

.type	four integers describing the axis function type fun (see list with xfopt('_datatype')), length, force and temperature unit exponents
.label	a string label for the axis
.unit	a string for the unit of the axis

xfopt _FieldType returns the current list of field types.

Response data

Response data sets correspond to sets of universal files of type 58 and are characterized by the fields

.w	abscissa values
.xf	response data (one column per response)
.dof	characteristics of individual responses (one row per column in the response data as detailed below)
.fun	general data set options, contain [FunType DFormat NPoints XSpacing Xmin XStep ZValue] as detailed in the ufread section on file format 58.
.idopt	options used for identification related routines (see idopt)
.header	header (5 text lines with a maximum of 72 characters)
.x	abscissa description (see xfopt('_datatype'))
.yn	numerator description (see xfopt('_datatype'))
.yd	denominator description (see xfopt('_datatype'))
.z	third axis description (see xfopt('_datatype'))
.group	(optional) cell array containing DOF group names
.load	(optional) loading patterns used in the data set

While the xf format (w and xf fields, see xf page 40) contain all the information needed to characterize the system dynamics, other data is needed to know its physical meaning. The other database wrapper fields describe this information.

DOF/channel dependent options describe characteristics of particular responses of a MIMO data set. The dof field contains one row per response/DOF with the following information

```
[RespNodeID.RespDOFID ExciNodeID.ExciDOFID Address ...
 RespGroupID ExciGroupID FunID LoadCase ZaxisValue]
```

Standard *DOF definitions* of the form NodeID.DOFID are introduced in section 7.5. *Addresses* are integer numbers used to identify columns of xf matrices. Sensor / actuator *groups* are correspond to the group names given in the group field (this is really only supported by ufread). In the standard database wrapper XF, the default value for the .dof field is 'XFdof' so that the standard global variable XFdof is used.

The idopt field is used to point to identification options used on the data set. The usual value for this field is 'IDopt' so that the options stored in the standard global variable IDopt are used.

The Group field is used to associate a name to the group identification numbers RespGroupID ExciGroupID defined in the .dof columns 4 and 5. These names are saved by ufwrite but currently not used in other parts of the *SDT*.

The load field describes *loading cases* by giving addresses of applied loads in odd columns and the corresponding coefficients in even columns. This field is used in test cases with multiple correlated inputs.

Shapes at DOFs

Shapes at DOFs is used to store modeshapes, time responses defined at all nodes, ... and are written to universal file format 55 (response at nodes) by ufwrite. The fields used for such datasets are

.po	pole values, time steps, frequency values ... For poles, see ii_pof which allows conversions between the different pole formats.
.res	residues / shapes (one row per shape). Residue format is detailed in section 2.8.
.dof	characteristics of individual responses (one row per column in the shape data). This field has the same format as response data .dof fields described above.
.fun	function characteristics (see ufread type 58)
.header	header (5 text lines with a maximum of 72 characters)
.idopt	identification options. This is filled when the data structure is obtained as the result of an idcom call.
.label	string describing the content
.lab_in	optional cell array of names for the inputs
.lab_out	optional cell array of names for the outputs
.group	optional cell group names

FEM / wire-frame model

FEM / wire-frame models are stored using at least the fields .Node and .Elt. For a complete list of possible FEM data structure fields see section 7.6. You can visualize the associated model using calls of the form

```
cf=feplot;cf.model={XF(1).Node,XF(1).Elt}
```

See also idopt, id_rm, iiplot, ufread

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] E. Balmes, “Orthogonal maximum sequence sensor placements algorithms for modal tests, expansion and visibility.,” *IMAC*, January 2005.

[3] E. Balmes, “Model reduction for systems with frequency dependent damping properties,” *International Modal Analysis Conference*, pp. 223–229, 1997.

[4] T. Hasselman, “Modal coupling in lightly damped structures,” *AIAA Journal*, vol. 14, no. 11, pp. 1627–1628, 1976.

[5] E. Balmes, “New results on the identification of normal modes from experimental complex modes,” *Mechanical Systems and Signal Processing*, vol. 10, no. 6, 1996.

[6] A. Plouin and E. Balmes, “A test validated model of plates with constrained viscoelastic materials,” *International Modal Analysis Conference*, pp. 194–200, 1999.

[7] E. Balmes and S. Germès, “Tools for viscoelastic damping treatment design. application to an automotive floor panel.,” *ISMA*, September 2002.

[8] K. McConnell, *Vibration Testing. Theory and Practice.* Wiley Interscience, New-York, 1995.

[9] W. Heylen, S. Lammens, and P. Sas, *Modal Analysis Theory and Testing.* KUL Press, Leuven, Belgium, 1997.

[10] “Vibration and shock - experimental determination of mechanical mobility,” *ISO 7626*, 1986.

[11] D. Ewins, *Modal Testing: Theory and Practice.* John Wiley and Sons, Inc., New York, NY, 1984.

[12] E. Balmes, "Integration of existing methods and user knowledge in a mimo identification algorithm for structures with high modal densities," *International Modal Analysis Conference*, pp. 613–619, 1993.

[13] E. Balmes, "Frequency domain identification of structural dynamics using the pole/residue parametrization," *International Modal Analysis Conference*, pp. 540–546, 1996.

[14] P. Guillaume, R. Pintelon, and J. Schoukens, "Parametric identification of multivariable systems in the frequency domain : a survey," *International Seminar on Modal Analysis, Leuven, September*, pp. 1069–1080, 1996.

[15] R. J. Craig, A. Kurdila, and H. Kim, "State-space formulation of multi-shaker modal analysis," *Int. J. Anal. and Exp. Modal Analysis*, vol. 5, no. 3, 1990.

[16] M. Richardson and D. Formenti, "Global curve fitting of frequency response measurements using the rational fraction polynomial method," *International Modal Analysis Conference*, pp. 390–397, 1985.

[17] A. Sestieri and S. Ibrahim, "Analysis of errors and approximations in the use of modal coordinates," *Journal of sound and vibration*, vol. 177, no. 2, pp. 145–157, 1994.

[18] E. Balmes, "Sensors, degrees of freedom, and generalized modeshape expansion methods," *International Modal Analysis Conference*, pp. 628–634, 1999.

[19] D. Kammer, "Effect of model error on sensor placement for on-orbit modal identification of large space structures," *J. Guidance, Control, and Dynamics*, vol. 15, no. 2, pp. 334–341, 1992.

[20] E. Balmes, "Review and evaluation of shape expansion methods," *International Modal Analysis Conference*, pp. 555–561, 2000.

[21] A. Chouaki, P. Ladevèze, and L. Proslier, "Updating Structural Dynamic Models with Emphasis on the Damping Properties," *AIAA Journal*, vol. 36, pp. 1094–1099, June 1998.

[22] E. Balmes, "Optimal ritz vectors for component mode synthesis using the singular value decomposition," *AIAA Journal*, vol. 34, no. 6, pp. 1256–1260, 1996.

[23] D. Kammer, “Test-analysis model development using an exact modal reduction,” *International Journal of Analytical and Experimental Modal Analysis*, pp. 174–179, 1987.

[24] J. O’Callahan, P. Avitabile, and R. Riemer, “System equivalent reduction expansion process (serep),” *IMAC VII*, pp. 29–37, 1989.

[25] R. Guyan, “Reduction of mass and stiffness matrices,” *AIAA Journal*, vol. 3, p. 380, 1965.

[26] R. Kidder, “Reduction of structural frequency equations,” *AIAA Journal*, vol. 11, no. 6, 1973.

[27] M. Paz, “Dynamic condensation,” *AIAA Journal*, vol. 22, no. 5, pp. 724–727, 1984.

[28] M. Levine-West, A. Kissil, and M. Milman, “Evaluation of mode shape expansion techniques on the micro-precision interferometer truss,” *International Modal Analysis Conference*, pp. 212–218, 1994.

[29] E. Balmes and L. Billet, “Using expansion and interface reduction to enhance structural modification methods,” *International Modal Analysis Conference*, February 2001.

[30] MSC/NASTRAN, *Quick Reference Guide 70.7.* MacNeal Shwendler Corp., Los Angeles, CA, February,, 1998.

[31] A. Girard, “Modal effective mass models in structural dynamics,” *International Modal Analysis Conference*, pp. 45–50, 1991.

[32] R. J. Craig, “A review of time-domain and frequency domain component mode synthesis methods,” *Int. J. Anal. and Exp. Modal Analysis*, vol. 2, no. 2, pp. 59–72, 1987.

[33] 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.

[34] C. Farhat and M. Géradin, “On the general solution by a direct method of a large-scale singular system of linear equations: Application to the analysis of floating structures,” *International Journal for Numerical Methods in Engineering*, vol. 41, pp. 675–696, 1998.

[35] R. J. Craig and M. Bampton, “Coupling of substructures for dynamic analyses,” *AIAA Journal*, vol. 6, no. 7, pp. 1313–1319, 1968.

[36] E. Balmes, “Use of generalized interface degrees of freedom in component mode synthesis,” *International Modal Analysis Conference*, pp. 204–210, 1996.

[37] E. Balmes, “Parametric families of reduced finite element models. theory and applications,” *Mechanical Systems and Signal Processing*, vol. 10, no. 4, pp. 381–394, 1996.

[38] E. Balmes, “Efficient sensitivity analysis based on finite element model reduction,” *International Modal Analysis Conference*, pp. 1168–1174, 1998.

[39] E. Balmes, “Super-element representations of a model with frequency dependent properties,” *International Seminar on Modal Analysis, Leuven, September*, vol. 3, pp. 1767–1778, 1996.

[40] T. Hughes, *The Finite Element Method, Linear Static and Dynamic Finite Element Analysis.* Prentice-Hall International, 1987.

[41] H. J.-P. Morand and R. Ohayon, *Fluid Structure Interaction.* J. Wiley & Sons 1995, Masson, 1992.

[42] 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.

[43] R. G. and V. C., “Calcul modal par sous-structuration classique et cyclique,” *Code_Aster, Version 5.0, R4.06.02-B*, pp. 1–34, 1998.

[44] S. Smith and C. Beattie, “Simultaneous expansion and orthogonalization of measured modes for structure identification,” *Dynamics Specialist Conference, AIAA-90-1218-CP*, pp. 261–270, 1990.

[45] C. Johnson, “Discontinuous galerkin finite element methods for second order hyperbolic problems,” *Computer methods in Applied Mechanics and Engineering*, no. 107, pp. 117–129, 1993.

[46] M. Hulbert and T. Hughes, “Space-time finite element methods for second-order hyperbolic equations,” *Computer methods in Applied Mechanics and Engineering*, no. 84, pp. 327–348, 1990.

[47] R. J. Craig and M. Blair, “A generalized multiple-input, multiple-ouptut modal parameter estimation algorithm,” *AIAA Journal*, vol. 23, no. 6, pp. 931–937, 1985.

[48] N. Lieven and D. Ewins, “Spatial correlation of modeshapes, the coordinate modal assurance criterion (comac),” *International Modal Analysis Conference*, 1988.

[49] D. Hunt, “Application of an enhanced coordinate modal assurance criterion,” *International Modal Analysis Conference*, pp. 66–71, 1992.

[50] R. Williams, J. Crowley, and H. Vold, “The multivariate mode indicator function in modal analysis,” *International Modal Analysis Conference*, pp. 66–70, 1985.

[51] E. Balmes, C. Chapelier, P. Lubrina, and P. Fargette, “An evaluation of modal testing results based on the force appropriation method,” *International Modal Analysis Conference*, pp. 47–53, 1995.

[52] A. W. Phillips, R. J. Allemang, and W. A. Fladung, *The Complex Mode Indicator Function (CMIF) as a parameter estimation method*. International Modal Analysis Conference, 1998.

[53] J. Imbert, *Analyse des Structures par Eléments Finis.* E.N.S.A.E. Cépaques Editions.

Index