Structural Dynamics Toolbox & FEMLink

For Use with MATLAB®

User's Guide

February 24, 2019

SDTools

How to Contact SDTools

33 +1 44 24 63 71	Phone
SDTools	Mail
44 rue Vergniaud	
75013 Paris (France)	

http://www.sdtools.com	Web
http://support.sdtools.com	Technical support
info@sdtools.com	Sales, pricing, and general information

Structural Dynamics Toolbox User's Guide on February 24, 2019

© Copyright 1991-2019 by SDTools

The software described in this document is furnished under a license agreement.
The software may be used or copied only under the terms of the license agreement.
No part of this manual in its paper, PDF and HTML versions may be copied, printed, or reproduced in any form without prior written consent from SDTools.

Structural Dynamics Toolbox is a registered trademark of SDTools
OpenFEM is a registered trademark of INRIA and SDTools
MATLAB is a registered trademark of The MathWorks, Inc.
Other products or brand names are trademarks or registered trademarks of their respective holders.

Contents

1 Preface **11**

1.1 Key areas . . . 12
1.2 Key notions in SDT architecture . . . 14
1.3 Typesetting conventions and scientific notations . . . 17
1.4 Other toolboxes from SDTools . . . 19
1.5 Licensing utilities . . . 20
1.5.1 Node locked licenses . . . 21
1.5.2 Floating licenses . . . 21
1.6 Release notes for SDT and FEMLink 7.1 . . . 23
1.6.1 Key features . . . 23
1.6.2 Detail by function . . . 24
1.6.3 Notes by MATLAB release . . . 26
1.7 Release notes for SDT and FEMLink 7.0 . . . 27
1.7.1 Key features . . . 27
1.7.2 Detail by function . . . 28
1.7.3 Notes by MATLAB release . . . 30
1.8 Release notes for SDT and FEMLink 6.8 . . . 31
1.8.1 Key features . . . 31
1.8.2 Detail by function . . . 32
1.8.3 Notes by MATLAB release . . . 34
1.9 Release notes for SDT and FEMLink 6.7 . . . 35
1.9.1 Key features . . . 35
1.9.2 Detail by function . . . 36
1.9.3 Notes by MATLAB release . . . 37

2 Modal test tutorial **39**

2.1 iiplot figure tutorial . . . 42
2.1.1 The main figure . . . 43
2.1.2 The curve stack . . . 46

2.1.3 Handling what you display, axes and channel tabs . . . 48
2.1.4 Channel tab usage . . . 49
2.1.5 Handling displayed units and labels . . . 50
2.1.6 SDT 5 compatibility . . . 50
2.1.7 iiplot for signal processing . . . 52
2.1.8 iiplot FAQ . . . 53
2.2 Identification of modal properties (Id dock) . . . 54
2.2.1 Opening and description of used data . . . 54
2.2.2 General process . . . 59
2.2.3 Importing FRF data . . . 63
2.2.4 Write a script to build a transfer structure . . . 65
2.2.5 Data acquisition . . . 66
2.3 Pole initialization (IdAlt and IdMain filling) . . . 67
2.3.1 External pole estimation . . . 67
2.3.2 LSCF . . . 68
2.3.3 Single pole estimate . . . 74
2.3.4 Band to pole estimate . . . 76
2.3.5 Direct system parameter identification algorithm . . . 76
2.3.6 Orthogonal polynomial identification algorithm . . . 77
2.4 Identification options . . . 77
2.5 Estimate shapes from poles . . . 79
2.5.1 Broadband, narrowband, ... selecting the strategy . . . 79
2.5.2 `Qual`: Estimation of pole and shape quality . . . 82
2.5.3 When `id_rc` fails . . . 88
2.6 Update poles . . . 91
2.6.1 Eup : for a clean measurement with multiple poles . . . 91
2.6.2 Eopt : for a band with few poles . . . 92
2.6.3 EupSeq and EoptSeq : sequential narrowband pole updating . . . 93
2.6.4 Example for practice . . . 93
2.6.5 Background theory . . . 96
2.7 Display shapes : geometry declaration, pre-test . . . 97
2.7.1 Modal test geometry declaration . . . 97
2.7.2 Sensor/shaker configurations . . . 99
2.7.3 Animating test data, operational deflection shapes . . . 101
2.8 MIMO, Reciprocity, State-space, ... 103
2.8.1 Multiplicity (minimal state-space model) . . . 103
2.8.2 Reciprocal models of structures . . . 105
2.8.3 Normal mode form . . . 107

3 Test/analysis correlation tutorial 111

3.1 Topology correlation and test preparation . . . 113
3.1.1 Defining sensors in the FEM model : data handling . . . 114
3.1.2 Test and FEM coordinate systems . . . 117
3.1.3 Sensor/shaker placement . . . 120
3.2 Test/analysis correlation . . . 121
3.2.1 Shape based criteria . . . 121
3.2.2 Energy based criteria . . . 127
3.2.3 Correlation of FRFs . . . 128
3.3 Expansion methods . . . 129
3.3.1 Underlying theory for expansion methods . . . 130
3.3.2 Basic interpolation methods for unmeasured DOFs . . . 131
3.3.3 Subspace based expansion methods . . . 132
3.3.4 Model based expansion methods . . . 134
3.4 Structural dynamic modification . . . 134

4 FEM tutorial **137**
4.1 FE mesh declaration . . . 139
4.1.1 Direct declaration of geometry (truss example) . . . 139
4.2 Building models with feutil . . . 140
4.3 Building models with femesh . . . 144
4.3.1 Automated meshing capabilities . . . 146
4.3.2 Importing models from other codes . . . 146
4.3.3 Importing model matrices from other codes . . . 147
4.4 The feplot interface . . . 149
4.4.1 The main feplot figure . . . 149
4.4.2 Viewing stack entries . . . 153
4.4.3 Pointers to the figure and the model . . . 153
4.4.4 The property figure . . . 153
4.4.5 GUI based mesh editing . . . 155
4.4.6 Viewing shapes . . . 156
4.4.7 Viewing property colors . . . 158
4.4.8 Viewing colors at nodes . . . 159
4.4.9 Viewing colors at elements . . . 159
4.4.10 feplot FAQ . . . 160
4.5 Other information needed to specify a problem . . . 162
4.5.1 Material and element properties . . . 162
4.5.2 Other information stored in the stack . . . 164
4.5.3 Cases GUI . . . 164
4.5.4 Boundary conditions and constraints . . . 166
4.5.5 Loads . . . 167

4.6 Sensors . . . 168
4.6.1 Sensor GUI, a simple example . . . 168
4.6.2 Sensor definition from a cell array . . . 169
4.6.3 Sensor data structure and init commands . . . 173
4.6.4 Topology correlation and observation matrix . . . 180
4.7 Stress observation . . . 184
4.7.1 Building view mesh . . . 184
4.7.2 Building and using a selection for stress observation . . . 186
4.7.3 Observing resultant fields . . . 187
4.8 Computing/post-processing the response . . . 187
4.8.1 Simulate GUI . . . 187
4.8.2 Static responses . . . 188
4.8.3 Normal modes (partial eigenvalue solution) . . . 189
4.8.4 State space and other modal models . . . 190
4.8.5 Viewing shapes, stress, energy, ... 192
4.8.6 Time computation . . . 192
4.8.7 Manipulating large finite element models . . . 194
4.8.8 Optimized assembly strategies . . . 196

5 Structural dynamic concepts 199
5.1 I/O shape matrices . . . 200
5.2 Normal mode models . . . 202
5.3 Damping . . . 203
5.3.1 Viscous damping in the normal mode model form . . . 203
5.3.2 Viscous damping in finite element models . . . 205
5.3.3 Hysteretic damping in finite element models . . . 206
5.4 State space models . . . 209
5.5 Complex mode models . . . 210
5.6 Pole/residue models . . . 212
5.7 Parametric transfer function . . . 214
5.8 Non-parametric transfer function . . . 214

6 Advanced FEM tools 217
6.1 FEM problem formulations . . . 219
6.1.1 3D elasticity . . . 219
6.1.2 2D elasticity . . . 220
6.1.3 Acoustics . . . 221
6.1.4 Classical lamination theory . . . 222
6.1.5 Piezo-electric volumes . . . 225
6.1.6 Piezo-electric shells . . . 227

6.1.7 Geometric non-linearity . . . 229
6.1.8 Thermal pre-stress . . . 231
6.1.9 Hyperelasticity . . . 231
6.1.10 Gyroscopic effects . . . 233
6.1.11 Centrifugal follower forces . . . 234
6.1.12 Poroelastic materials . . . 234
6.1.13 Heat equation . . . 239
6.2 Model reduction theory . . . 241
6.2.1 General framework . . . 241
6.2.2 Normal mode models . . . 242
6.2.3 Static correction to normal mode models . . . 244
6.2.4 Static correction with rigid body modes . . . 245
6.2.5 Other standard reduction bases . . . 246
6.2.6 Substructuring . . . 247
6.2.7 Reduction for parameterized problems . . . 249
6.3 Superelements and CMS . . . 250
6.3.1 Superelements in a model . . . 250
6.3.2 SE data structure reference . . . 251
6.3.3 An example of SE use for CMS . . . 253
6.3.4 Obsolete superelement information . . . 255
6.3.5 Sensors and superelements . . . 256
6.4 Model parameterization . . . 258
6.4.1 Parametric models, zCoef . . . 258
6.4.2 Reduced parametric models . . . 261
6.4.3 upcom parameterization for full order models . . . 261
6.4.4 Getting started with upcom . . . 262
6.4.5 Reduction for variable models . . . 263
6.4.6 Predictions of the response using upcom . . . 264
6.5 Finite element model updating . . . 265
6.5.1 Error localization/parameter selection . . . 266
6.5.2 Update based on frequencies . . . 267
6.5.3 Update based on FRF . . . 267
6.6 Handling models with piezoelectric materials . . . 269
6.7 Viscoelastic modeling tools . . . 269
6.8 SDT Rotor . . . 269

7 Developer information **271**
7.1 Nodes . . . 273
7.1.1 Node matrix . . . 273
7.2 Model description matrices . . . 274

7.3 Material property matrices and stack entries . . . 276
7.4 Element property matrices and stack entries . . . 277
7.5 DOF definition vector . . . 278
7.6 FEM model structure . . . 280
7.7 FEM stack and case entries . . . 281
7.8 FEM result data structure . . . 285
7.9 Curves and data sets . . . 286
7.10 DOF selection . . . 292
7.11 Node selection . . . 293
7.12 Element selection . . . 296
7.13 Defining fields trough tables, expressions, ... 299
7.14 Constraint and fixed boundary condition handling . . . 301
7.14.1 Theory and basic example . . . 301
7.14.2 Local coordinates . . . 302
7.14.3 Enforced displacement . . . 303
7.14.4 Resolution as MPC and penalization transformation . . . 303
7.14.5 Low level examples . . . 304
7.15 Internal data structure reference . . . 305
7.15.1 Element functions and C functionality . . . 305
7.15.2 Standard names in assembly routines . . . 306
7.15.3 Case.GroupInfo cell array . . . 308
7.15.4 Element constants data structure . . . 309
7.16 Creating new elements (advanced tutorial) . . . 311
7.16.1 Generic compiled linear and non-linear elements . . . 311
7.16.2 What is done in the element function . . . 312
7.16.3 What is done in the property function . . . 313
7.16.4 Compiled element families in of_mk . . . 315
7.16.5 Non-linear iterations, what is done in of_mk . . . 320
7.16.6 Element function command reference . . . 321
7.17 Variable names and programming rules (syntax) . . . 327
7.17.1 Variable naming conventions . . . 328
7.17.2 Coding style . . . 329
7.17.3 Input parsing conventions . . . 331
7.17.4 Commands associated to project application functions . . . 332
7.17.5 Commands associated to tutorials . . . 336
7.18 Criteria with CritFcn . . . 338
7.19 Legacy information . . . 338
7.19.1 Legacy 2D elements . . . 339
7.19.2 Rules for elements in of_mk_subs . . . 339

8 GUI and reporting tools **347**
8.1 Formatting MATLAB graphics and output figures . . . 348
8.1.1 Formatting operations with `objSet` . . . 349
8.1.2 Persistent data in `Project` . . . 349
8.1.3 `OsDic` dictionnary of names styles . . . 350
8.1.4 File name generation with `objString` . . . 352
8.1.5 Image generation with `ImWrite` . . . 352
8.2 SDT Tabs . . . 352
8.2.1 Project . . . 353
8.2.2 FEMLink . . . 353
8.2.3 Mode . . . 356
8.2.4 TestBas : position test versus FEM . . . 359
8.2.5 StabD : stabilization diagram . . . 362
8.2.6 Ident : pole tuning . . . 364
8.2.7 MAC : Modal Assurance Criterion display . . . 365
8.3 Non java SDT Tabs (obsolete) . . . 367
8.4 Handling data in the GUI format . . . 367
8.4.1 Parameter/button structure . . . 367
8.4.2 DefBut : parameter/button defaults . . . 369
8.4.3 Reference button file in CSV format . . . 370
8.4.4 Data storage and access . . . 371
8.4.5 Tweaking display . . . 374
8.4.6 Defining an exploration tree . . . 376
8.4.7 Finding `CinCell` buttons in the GUI with `getCell` . . . 377
8.5 Interactivity . . . 378
8.5.1 Handling tabs . . . 378
8.5.2 Handling dependencies . . . 378
8.5.3 Dialogs . . . 380

9 Element reference **383**
bar1 . . . 386
beam1, beam1t . . . 387
celas,cbush . . . 390
dktp . . . 394
fsc . . . 396
hexa8, penta6, tetra4, and other 3D volumes . . . 399
integrules . . . 400
mass1,mass2 . . . 409
m_elastic . . . 410
m_heat . . . 414

m_hyper . . . 416
m_piezo . . . 418
p_beam . . . 420
p_heat . . . 424
p_shell . . . 428
p_solid . . . 433
p_spring . . . 436
p_super . . . 438
p_piezo . . . 440
quad4, quadb, mitc4 . . . 444
q4p, q8p, t3p, t6p and other 2D volumes . . . 447
rigid . . . 448
tria3, tria6 . . . 451

10 Function reference **453**
abaqus . . . 460
ans2sdt . . . 471
basis . . . 475
comgui,cingui . . . 479
commode . . . 489
comstr . . . 491
curvemodel . . . 495
db, phaseb . . . 497
ex2sdt . . . 498
fe2ss . . . 501
fecom . . . 505
femesh . . . 524
feutil . . . 538
feutila . . . 570
feutilb . . . 571
feplot . . . 589
fesuper . . . 594
fjlock . . . 603
fe_c . . . 609
fe_case . . . 612
fe_caseg . . . 627
fe_ceig . . . 631
fe_coor . . . 633
fe_curve . . . 635
fe_cyclic . . . 643

fe_def . . . 647
fe_eig . . . 650
fe_exp . . . 654
fe_gmsh . . . 658
fe_load . . . 663
fe_mat . . . 668
fe_mknl, fe_mk . . . 671
fe_norm . . . 677
fe_quality . . . 679
fe_range . . . 685
fe_reduc . . . 698
fe_sens . . . 703
fe_simul . . . 709
fe_stress . . . 711
fe_time . . . 716
of_time . . . 728
idcom . . . 732
idopt . . . 737
id_dspi . . . 740
id_nor . . . 742
id_poly . . . 746
id_rc, id_rcopt . . . 747
id_rm . . . 751
iicom . . . 754
iimouse . . . 766
iiplot . . . 770
ii_cost . . . 774
ii_mac . . . 775
ii_mmif . . . 787
ii_plp . . . 794
ii_poest . . . 799
ii_pof . . . 801
lsutil . . . 803
nasread . . . 805
naswrite . . . 810
nor2res, nor2ss, nor2xf . . . 816
of2vtk . . . 824
ofact . . . 826
perm2sdt . . . 831

polytec . . . 833
psi2nor . . . 836
qbode . . . 838
res2nor . . . 840
res2ss, ss2res . . . 841
res2tf, res2xf . . . 844
rms . . . 845
samcef . . . 846
sd_pref . . . 848
setlines . . . 849
sdtcheck . . . 850
sdtdef . . . 852
sdth . . . 855
sdthdf . . . 856
sdtroot, sdt_locale . . . 861
sdtweb . . . 864
sp_util . . . 866
stack_get,stack_set,stack_rm . . . 868
ufread . . . 870
ufwrite . . . 876
upcom . . . 878
up_freq, up_ifreq . . . 887
up_ixf . . . 888
v_handle . . . 889
xfopt . . . 890

Bibliography **893**

Index **898**

Preface

1.1 Key areas 12
1.2 Key notions in SDT architecture 14
1.3 Typesetting conventions and scientific notations 17
1.4 Other toolboxes from SDTools 19
1.5 Licensing utilities 20
1.5.1 Node locked licenses 21
1.5.2 Floating licenses 21
1.6 Release notes for SDT and FEMLink 7.1 23
1.6.1 Key features 23
1.6.2 Detail by function 24
1.6.3 Notes by MATLAB release 26
1.7 Release notes for SDT and FEMLink 7.0 27
1.7.1 Key features 27
1.7.2 Detail by function 28
1.7.3 Notes by MATLAB release 30
1.8 Release notes for SDT and FEMLink 6.8 31
1.8.1 Key features 31
1.8.2 Detail by function 32
1.8.3 Notes by MATLAB release 34
1.9 Release notes for SDT and FEMLink 6.7 35
1.9.1 Key features 35
1.9.2 Detail by function 36
1.9.3 Notes by MATLAB release 37

1.1 Key areas

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.

You can find a primer for beginners at `http://www.sdtools.com/help/primer.pdf`.

Self contained code examples are distributed throughout the manual. Additional demonstration scripts can be found in the `sdt/sdtdemos` directory which for a proper installation should be in your MATLAB path. If not, use `sdtcheck path` to fix your path.

The MATLAB `doc` command no longer supports non MathWorks toolboxes, documentation access is thus now obtained with `sdtweb FunctionName`.

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, followed parametric identification) with information about the spatial position of multiple sensors and actuators.

An experimental modal analysis project can be decomposed in following steps

- before the test, preparation and design (see section 2.7)
- acquisition of test data, import into the SDT, direct exploitation of measurements (visualization, operational deflection shapes, ...) (see section 2.1)
- identification of modal properties from test data (see section 2.2)
- handling of MIMO tests and other model transformations (output of identified models to state-space, normal mode, ... formats, taking reciprocity into account, ...) (see section 2.8)

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.

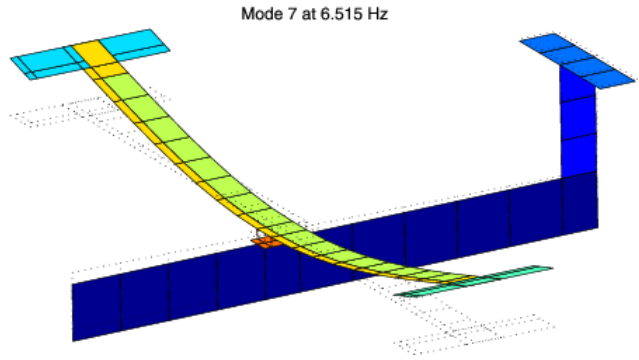


Figure 1.1: GARTEUR structure.

- `gartfe` builds the finite element model using the `femesh` pre-processor
- `gartte` shows how to prepare the visualization of test results and perform basic correlation
- `gartid` does the identification on a real data set
- `d_cor('TutoSensPlace')` discusses sensor/shaker placement

Area 2: Test/analysis correlation

Correlation between test results and finite element predictions is a usual motivation for modal tests. Chapter 3 addresses topology correlation, test preparation, correlation criteria, modeshape expansion, and structural dynamic modification. Details on the complete range of sensor definitions supported by SDT can be found in 4.6. Indications on how to use *SDT* for model updating are given in section 6.5 .

- `gartco` shows how to use `fe_sens` and `fe_exp` to perform modeshape expansion and more advanced correlation
- `gartup` shows how the `upcom` interface can be used to further correlate/update the model

Area 3: Basic finite element analysis

Chapter 4 gives a tutorial on FEM modeling in *SDT*. Developer information is given in chapter 7. Available elements are listed in chapter 9.

A good part of the finite element analysis capabilities of the *SDT* are developed as part of the OpenFEM project. OpenFEM is typically meant for developers willing to invest in a stiff learning curve but needing an Open Source environment. *SDT* provides an integrated and optimized access to OpenFEM and extends the library with

- solvers for structural dynamics problems (eigenvalue (`fe_eig`), component mode synthesis (section 6.3), state-space model building (`fe2ss`), ... (see `fe_simul`);
- 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 (see section 4.4);
- high level handling of FEM solutions using cases;
- interface with other finite element codes through the FEMLink extension to *SDT*.

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 6.2 . Typical applications are treated in section 6.3 .

Finally, as shown in section 6.4 , the *SDT* supports many tools necessary for finite element model updating.

1.2 Key notions in SDT architecture

`functions, commands`

To limit the number of functions SDT heavily relies on the use of string `commands`. Functions group related commands (`feutil` for mesh manipulation, `iiplot` for curve visualization, ...). Within each functions commands (for example `iicom ImWrite`), are listed with their options.

command string and structure options (CAM,Cam,RO)

Most SDT functions accept inputs of the form `function('command',data, ...)`.

Command **options** can be specified within the command (parsed from the string). `iicom('ch+5')` is thus parsed to ask for a step of +5 channels. See `commode` for conventions linked to parsed commands (case insensitive, ...).

When reading SDT source code, look for the `CAM` (original command) and `Cam` (lower case version of the command). Section 7.17 gives more details on SDT coding style.

While command parsing is very often convenient, it many become difficult to use in graphical user interfaces or when to many options are required. SDT thus typically supports a mechanism to provide options using either commands options, or option values as a data structure typically called `RO` (for **Run Options** but any variable name is acceptable). Support for both string and structure options is documented and is being generalized to many commands.

```
% Equivalent command an structure calls
figure(1);plot(sin(1:10));title('Test');legend('sin');
cd(sdtdef('tempdir')); % Use SDT temp dir

% Give options in string
comgui('ImWrite -NoCrop Test.png')
% Give options as structure (here allows dynamic generation of title)
RO=struct('NoCrop',1,'FileName',{{pwd,'@Title','@legend','.png'}});
comgui('ImWrite',RO);
```

structures used for typical data

The SDT supports a number of data structures used to store common structures. The main structures are

- `model` for FEM models and wire frame displays
- `def` for responses at DOF
- `curve` for multi-dimensional data
- `sens` sensor definition, see section 4.6.3 .

Stack

When extensible and possibly large lists of mixed data are needed, SDT uses `.Stack` fields which are N by 3 cell arrays with each row of the form `{'type','name',val}`. The purpose of these cell arrays is to deal with unordered sets of data entries which can be classified by type and name.

`stack_get`, `stack_set` and `stack_rm` are low level functions used to get/set/remove single or multiple entries from stacks.

Higher level pointer access to stacks stored in `iiplot` (curve stacks) and `feplot` (model and case stacks) are described in section 2.1.2 and section 4.5.3 .

GUI Graphical User Interfaces

GUI functions automatically generate views of data and associated parameters. The main GUI in SDT are

- `iiplot` and the associated `iicom` (commands to edit plots) to view frequency and time responses defined at multiple channels.
- `feplot` and the associated `fecom` (commands to edit plots) to view 3D FEM and test meshes and responses.
- `idcom` for experimental modal analysis.
- `ii_mac` for test/analysis correlation.
- `sdtroot` for project handling, parameter editing.

Graphically supported operations (interactions between the user and plots/ menus/mouse movements/key pressed) are documented under `iimouse`.

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.

pointers (and global variables)

Common data is preferably stored in the `userdata` of graphical objects. SDT provides two object types to ease the use of `userdata` for information that the user is likely to modify

- *SDT* handle objects implement methods used to access data in the feplot figure (see section 4.4.3), the iiplot figure (see section 2.1.2), or the ii_mac menu.
- v_handle to allow editing of user data of any userdata.

For example in a feplot figure, cf=feplot(5) retrieves the *SDT* handle object associated with the figure, while cf.mdl is a *SDT* handle method that retrieves the v_handle object where the model data structure is stored.

global variables are no longer used by SDT, since that can easily be source of errors. The only exceptions are upcom which will use the global variable Up if a model is not provided as argument and the femesh user interface for finite element mesh handling (feutilimplements the same commands without use of global variables), which uses the 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 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, ... 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`	blue monospace font : Matlab function names, variables
`feplot`	light blue monospace font: SDT function names
`command`	pink : strings and SDT `Commands`
var	italic pink: part of command strings that have to be replaced by their value
`% comment`	green: comments in script examples
Italics	MATLAB Toolbox names, mathematical notations, and new terms when they are defined
Bold	key names, menu names and items
Small print	comments
`(1,2)`	the element of indices `1`, `2` of a matrix
`(1,:)`	the first row of a matrix
`(1,3:end)`	elements `3` to whatever is consistent of the first row of a matrix

Programming rules are detailed under section 7.17 . Conventions used to specify string commands used by user interface functions are detailed under `commode`.

Usual abbreviations are

CMS	Component Mode Synthesis (see section 6.3.3)
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 5.1). $\left\{\phi_j^T b\right\}, \left\{\psi_j^T b\right\}$ modal input matrix of the j^{th} normal / complex mode
$[c]$	sensor output shape matrix, model with N DOFs and NS outputs (see section 5.1). $\{c\phi_j\}, \{c\psi_j\}$ modal output matrix of the j^{th} normal / complex mode
$[E]_{NS\times NA}$	correction matrix for high frequency modes (see section 5.6)
$[F]_{NS\times NA}$	correction matrix for low frequency modes (see section 5.6)
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 5.2)
$\{q\}_{N\times 1}$	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 5.6)
$[T_j]$	$= \{c\phi_j\}\left\{\phi_j^T b\right\}$ residue matrix of the j^{th} normal mode (used for proportionally damped models) (see section 5.6)
$\{u(s)\}_{NA\times 1}$	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 $[Ms^2 + Cs + K]$)
$[\alpha(s)]$	dynamic compliance matrix (force to displacement transfer function)
p, α	design parameters of a FE model (see section 6.4.2)
$\Delta M, \Delta C, \Delta K$	additive modifications of the mass, damping and stiffness matrices (see section 6.4.2)
$[\Gamma]$	non-diagonal modal damping matrix (see section 5.3)
λ_j	complex pole (see section 5.5)
$[\phi]_{N\times NM}$	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 5.2)
$[\theta]_{N\times NM}$	NM complex modes of a first order symmetric structural model (see section 5.5)
$[\psi]_{N\times NM}$	NM complex modes of damped structural model (see section 5.5)

1.4 Other toolboxes from SDTools

SDTools also develops other modules that are distributed under different licensing schemes. These

modules are often much less documented and address specialized themes, so that only a technical discussion of what you are trying to achieve will let us answer the question of whether the module is useful for you.

- Viscoelastic tools : an SDT extension for the analysis and design of viscoelastic damping. Beta documentation at `http://www.sdtools.com/help/visc.pdf`.
- Rotor tools : an SDT extension for rotor dynamics and cyclic symmetry. Beta documentation at `http://www.sdtools.com/help/rotor.pdf`.
- Contact tools : an SDT extension for contact/friction handling (generation observation matrices, tangent coupling matrices, various post-treatments). Beta documentation at `http://www.sdtools.com/help/contactm.pdf`.
- non linear vibration tools : an SDT extension for non-linear vibration and in particular time and frequency domain simulation of problems with contact and friction.
- OSCAR : a module for the study of pantograph/catenary interaction developed with SNCF.

Selected cross references to these other modules are listed here.

- `fevisco Range` this command is part of the viscoelastic tools.
- `fe2xf` this function is part of the viscoelastic tools.
- `fe_cyclicb ShaftEig` this command is part of the rotor tools.
- `Follow` is part of the contact and rotor tools. `nl_spring` is the generic implementation of time domain non-linearities in SDT.
- *ExtEq*`http://www.sdtools.com/help/eq_dyn.html#eq*ce_shell`
- *ExtEq*`http://www.sdtools.com/help/eq_dyn.html#eq*pze_c`
- *ExtEq*`http://www.sdtools.com/help/eq_dyn.html#Electrode`

1.5 Licensing utilities

1.5.1 Node locked licenses

For a node locked license. To obtain license information, download `https://www.sdtools.com/distrib/RLMmex.zip`.

- For windows, save the mex file in `sdt/804/sdtrlm.mexw64`.
- For Linux, save the mex file in `sdt/7.5/sdtrlm.mexw64`
- To obtain configuration information for license generation, at the MATLAB prompt use

```
which sdtrlm % to check the mex is visible (possibly rehash toolboxreset may be nee
sdtrlm hostid
```

- Send the associated information by email to `request@sdtools.com`, so that we can generate a license `sdt.lic` file. This will need to be saved in your MATLAB `prefdir` or in the directory where the `sdtrml` mex is located.
- You will need to restart MATLAB to access your license.

1.5.2 Floating licenses

Floating SDT licenses can use the RLM license manager. To install the server, download `https://www.sdtools.com/distrib/RLM.zip`.

- For windows, save the `RLM.zip/win64` directory to the target location of your server and start a shell (`cmd.exe`)
- For Linux, save the `RLM.zip/glnxa64` directory to the target location of your server and start a shell.
- Obtain configuration information for the license generation (note the second line will fail if you do not yet have a RLM server on that machine).

```
cd MyServerLocation
rlmutil rlmhostid
rlmutil rlmstat
```

- Send the associated information by email, so that we can generate a license `sdt.lic` file for your license server.

- Once you have received the `sdt.lic` file and placed it in the server directory where you will also find the `sdt.set` file. You can start the server using

  ```
  cd MyServerLocation
  rlm > outputfile
  rlmutil rlmhostid
  ```

  Note that you should NEVER run the RLM server as a priviledged user (root on unix or administrator on Windows). You can also find more administration help at `http://www.reprisesoftware.com/RLM_License_Administration.pdf`. In particular, the `-install_service` option is useful for windows, and boot time init is described for Linux.

On the client side (local copies of SDT), you will need to follow the procedure for SDT installation at `https://www.sdtools.com/faq/Release.html`, you must in particular provide the license request of the first installation matchine (further copies can then later be made as detailed below) and check that your SDT

- has the `sdtrlm` mex file. Use `which sdtrlm` to verify where this is located.
- have a license file `sdt.lic` located in the MATLAB preference directory (prefdir). `sdtkey('licfile')` returns the currently used location.
- That the license file is correct. Typically when a license server is used, the file only contains two lines `HOST` specifying the server name and port, `ISV sdt` specifying the use of an SDT server. The `port` specification on the second line may be necessary in configurations with firewalls but may be deleted otherwise.

  ```
  # type(fullfile(prefdir,'sdt.lic')) % for display in MATLAB
  HOST NameOfServer  ANY 5053
  ISV sdt port=50175
  ```

- To check the status of licenses used in your current MATLAB session use the following and possibly send the result to SDTools for diagnostic

  ```
  sdtcheck('rlm')
  ```

- For details on the server status `sdtcheck('rlmstat')`.
- Please note that for multiple installations, you simply need to use a network location (windows : windows server or Linux server with SAMBA, linux: NFS mount or equivalent) or copy the full SDT directory and possibly the license file `sdt.lic` to the user preference directory using

  ```
  copyfile(which('sdt.lic'),prefdir);
  ```

1.6 Release notes for SDT and FEMLink 7.1

1.6.1 Key features

SDT 7.1 is the only version fully compatible with MATLAB 9.4 (2018b) to 9.6 (2019a) mostly due to changes in the representation of complex numbers in MATLAB. Key changes of this release are

- A continued effort in making the experimental modal analysis part of SDT section 2.2 fully accessible without any script is nearly complete. Functions however obviously remain accessible from the command line to users will to learn how to use them. The associated docks `Id` (for experimental modal analysis see section 2.2), `CoTopo` (topology correlation see section 3.1) and `CoShape` (test/FEM correlation see section 3.2) have been extended and tutorials have been introduced.
- A major effort was put on the documentation. The new structuration of demos into tutorials helps training. You can for example see tutorials in various files with `d_mesh('tuto')`, `gartid('tuto')`, `d_cor('tuto')`, `d_cms('tuto')`, ... Equations are now shown as SVG files which improves readabily, but may pose problems on some older versions of MATLAB where the help browser does not support SVG.
- We are still working with the MathWorks on improving reliability of the help browser. To bypass some bugs, you may have to change default location where the help is shown using `sdtdef('browser-SetPref','-helpbrowser')` or `sdtdef('browser-SetPref','-webbrowser')`. For clickable areas of SVG figures, use Ctrl-Click to open in a new window or right-click and select Open in a new tab.

Outside improved robustness of the `femlink` GUI, key changes for FEMLink are

- `ans2sdt` extended BDF reading in particular for orthotropic materials and substructure export (to ease superelement import). Job submission integration is now supported as a consulting project feature.
- `nasread` compatibility with NX Nastran BGSET and BSURFS cards. Documentation of superelement (see `d_cms('TutoNasCb')`). Performance of `MAT9` and set reading.
- `abaqus` significant `.inp` reading improvements `*distribution,*hyperelastic`, set handling, ... Performance of large `.fil` reading. Robustness and performance enhancements of `resolve` commands. Introduction of a `.dat` reading framework for customer use, with complex modes output reading support.

For MATLAB compatibility see section 1.6.3 .

1.6.2 Detail by function

comgui	improved robustness and performance of Java interfaces, dock handling, menu_generation mechanism associated with OsDic.
demosdt	Tutorials underwent a major rewrite. d_cms now documents direct NASTRAN superelement import.
fe2ss	extended and improved documentation of damping handling strategies.
fecom	robustness of Show commands. In particular, ShowFi... now allows custom inits and is automatically added to the context menu.
fesuper	improvement of SE definition strategies with SEAdd, improved support of p_super definitions with SEinitCoef, and assembly calls with MatTyp -1 and -2. New command SeDofShow to display selected superelement active DOF on a full FE model in feplot.
feutil	Notable performance and generality improvements in the handling of sets. Support of pyra elements. Support of regular expression on sename searches selet eltname SE:#se[0-9]*, introduction of exclusion type in node and element selection operations. New operator &~ to subtract a selection from a current result. Introduction of element set exclusion using :exclude token following setname. Introduction of element selection type safesetname that returns empty elements instead of an exception. Support of setnames in double quotes for robust handling of setnames with special characters and spaces.
fe_caseg	Introduction of high level parametrization procedures for isotropic materials, any structural element and superelement, with command series Par*
fe_cyclic	improved support for multi-dimensional periodicity. This can be used with the support/fe_homo.m file which SDTools provides for free but with no support guarantee.
fe_eig	continued performance enhancements associated with memory management techniques, introduction of an Out-Of-Core modal basis storage support for method 5 (Lanczos).
fe_exp	MDRE expansion has been significantly enhanced and an initial version of an expansion tab is now provided.
fe_gmsh	introduced support for the new GMSH 4.0 format.
fe_mat	Improved robustness of unit conversion commands.
fe_mpc	Extended Rbe3 and CleanUsed commands.
fe_norm	major performance improvement of MSeq procedure. Introduction of an option to force vector collinearity tolerance estimation in the normalization procedure.

fe_range	Introduction of a Genetic algorithm framework with command GeneLoop. Introduction of an output data handling command Res that allows extracting and/or reformatting output data. Improvement of data sampler object getXFslice and introduction of an interpolation mode for coarse gridded data.
fe_reduc	continued performance enhancements associated with memory management techniques.
fe_shapeoptim	partially supported function for mesh morphing field projection is now included in the distribution.
fe_stress	Extended CritFcn calls, support of piezoelectric volumes, and export of weighted volumes associated with Gauss points in the .wjdet field.
mex	all SDT mex files now properly support the new complex number storage of MATLAB.
idcom	major improvement of band selection and pole extraction in stabilization diagrams. Improved dock functionality, performance and robustness. Menus for data manipulation (permute IO, SvdCur, ...) are introduced. Keyboard interaction has been improved. performance and robustness of the Channel tab has been improved.
id_rc	improvements of signal utilities dbsdt, filter, rms, a weights
iicom	improved file and dock reloading. Improved robustness of linked plots (magnitude/phase), keyboard interactions, java interaction.
ii_mac	the dockCoShape was notably extended and documented.
m_piezo	see sdtweb('pz_new') for specific release notes.
moldflow	This FEMLink function provides partial support of import of models exported by MoldFlow in Nastran, Universal and ANSYS formats.
polytec	improved translation of metadata associated with measurements.
p_shell	Merge commands have been extended for piezo applications.
p_solid	Support for element by element changes of properties has been notable extended.
pyra5	a new 5 node pyramid element is supported to ease mesh refining strategies in particular with level set strategies in lsutil.
sdtcheck	Utilities for sdtrootdir, rlmstat, rlm, patchfile were extended and robustified for use in patching and demos.
sdtroot	Subcommand @sfield for advanced struct manipulations is now supported.
sdtdef	clear definition of preferences with session scope (by default) or permanent scope (-setpref). Revision to alleviate preference file corruption with simultaneous statups. New commands envSet, envWrite to allow preferences load/ in .env files independently from the MATLAB session, compatible with deployed applications.

sdtacx	now supports section insertion in Word for easier report generation.
sdtweb	_tuto command provides generic support of tutorials the new base format for SDT demos.
comstr	robustness enhancements. -39 exports matlab variables to Python script. Support of nested string parsing with """ tokens in -25 calls.
mkl_utils	this mex file used to optimize time integration processes is now included in the base SDT.
ofact	sdtcheck(''patchMkl''') can be used to install the Pardiso solver which now supports complex matrices and can be notably faster for solutions with few right hand side solves. umfpack method is now properly supported for recent MATLAB.

1.6.3 Notes by MATLAB release

- MATLAB 8.0 (2012b) to 9.6 (2019a). *SDT & FEMLink 7.0* are developed for these versions of MATLAB and are fully compatible with them.
- For best performance, using MATLAB 9.0 (2016a) and higher is advised.
- For efficient FEM rendering, it is strongly advised to use HG2 : Matlab 8.4, R2014b and later.
- MATLAB 7.14 (2012a) to 8.3 (2014a) *SDT & FEMLink 7.0* are being phased out but can be used for a number of operations. Equations are not being shown correctly in the HTML documentation.
- Earlier MATLAB releases are no longer supported.
- MATLAB 8.5 has known bugs in the handling of colorbar.

1.7 Release notes for SDT and FEMLink 7.0

1.7.1 Key features

SDT 7.0 is the only version compatible with MATLAB 9.2 (2017a), 9.3 (2017b) and 9.4 (2018b) mostly due to ongoing improvements of MATLAB graphics. Key changes of this release are

- A full rewrite and major extension of modal analysis graphical interfaces and documentation detailed in section 2.2 . Step-by-step tutorials, such as section 2.2.1 , include buttons of the form which you can use to execute a step. LSCF and stabilization diagrams are now supported.
- The new notion of `docks` corresponds to MATLAB docks where multiple figures are combined for a typical use. Currently supported docks are
  - `Id` : for experimental modal analysis see section 2.2
  - `TestFEM` : topology correlation see section 3.1
  - `MAC` : test/FEM correlation see `ii_mac`.
- A major update of SDT GUI with most existing tabs ported to Java mode and necessary in docks. You can set the default tab to Java mode using `sdtdef('JavaUI',1)` or turn it off with `sdtdef('JavaUI',0)`. User documentation of tabs can be found in section 8.2 . Developer level documentation of GUI functions is now included in section 8 .
- Use `sdtweb('feplot','webbrowser')` to bypass the not yet fixed MATLAB bug where the links within pages are not called appropriately.

Key changes for FEMLink are

- `ans2sdt` improved import of `.cdb` and support of contacts.
- `nasread` Direct import of EXTESOUT output to SDT superelement format. Continued enhancements of `bulk` and `op2` reading. Initial support of `.op2` format writting of responses.
- `abaqus` continued enhancements of `.INP` reading in particular for composites and superelements, contact, ... Significant writing enhancements.
- GUI import of models is supported with the `FEMLink` tab, section 8.2.2 .

For MATLAB compatibility see section 1.7.3 .

1.7.2 Detail by function

This list is not yet complete.

basis	Clarified error for repeated BasId. New methods for multibody transformations.
cbush	major rewrite of documentation and introduced support commands for non-linear applications.
comgui	comgui gui command clarifies robust opening of feplot, iiplot figures linked to projects. Robustness in presence of mixed MATLAB/Java figures was improved.
fe_case	robustness enhancements in name matching. getFixDof implemented as subfunction to allow external calls.
fe_cyclic	compatibility with multi-physic periodic problems was enhanced.
fe_curve	Improved sweep generation and many minor improvements
fe_def	many detail improvements on silent operation and robustness. SubResample command implements optimized resampling.
fe_eig	tolerance strategy was changed in solver 5 for improved convergence.
fe_exp	underwent a major revision allowing the use of MDRE and MDRE-WE algorithms with the use of reduced models as well as proposing associated energy displays.
fe_load	Improved generality of DofSet case entries.
fe_mk	Improved support of orientation maps in particular for stress computations.
fe_quality	introduced a clean command to clean meshes in particular by straightening edges.
fe_range	major improvement of GUI operations and stat commands.
fe_reduc	Improved compatibility with parametric models, performance for large models, static correction in poorly conditionned cases.
fe_sens	Major rewrite of TestBas tab section 8.2.4 and associated commands.
	TdofTable now supports callbacks Distance view and SensorZoom selection.
FEMLink	GUI operation is now supported.
fe_stress	Corrected stress computation problems in bar1 elements.
fesuper	
	SeRestit was enhanced for restitution of multibody results.
id_rc	Robustness enhancements for QualTable, see section 2.5.2
id_rm	new commands such as PermuteIO, ...
ii_mac	Significant enhancements of GUI, context menus and docked operations.
iimouse	Robustness enhancements in particular for multi-figure interactivity used in docks
m_elastic	Continued extensions of orthotropic material support with orientation maps.
polytec	Official support of Polytec file access interfacing.
p_piezo	Bug correction in cases with material orientation.
p_beam	Revised handling of 3D section views.
sdtroot	Robustness enhancements for project handling and export to Word/PowerPoint. Evolution of Java tables with MATLAB changes.
ufread	Support of GUI operation.

1.7.3 Notes by MATLAB release

- MATLAB 8.0 (2012b) to 9.3 (2017b). *SDT & FEMLink 7.0* are developed for these versions of MATLAB and are fully compatible with them. Minor incompatibilities with 9.4 (2018a) are associated with the new complex number handling in MATLAB and and will be fixed with SDT 7.1.
- For best performance, using MATLAB 9.0 (2016a) and higher is advised.
- For efficient FEM rendering, it is strongly advised to use HG2 (Matlab 8.4, R2014b).
- MATLAB 7.6 (2008a) to 7.14 (2012a). *SDT & FEMLink 7.0* are being phased out but can be used for a number of operations.
- Earlier MATLAB releases are no longer supported.
- MATLAB 8.5 has known bugs in the handling of `colorbar`.

1.8 Release notes for SDT and FEMLink 6.8

1.8.1 Key features

SDT 6.8 is the only version compatible with MATLAB 8.6 (2015b) and 9.0 (2016a) mostly due to ongoing improvements of MATLAB graphics. Key changes of this release are

- A continued upgrade for SDT GUI with additional interactive java tabs. Image generation, described in section 8.1 , illustrates definition of projects `Project` for output file generation and handling of style sheets in `OsDic`. The java library `cinguj.jar` must be on the static path (check with `sdtcheck('PatchJavapath')`). Developer level documentation of GUI functions is now included in section 8 .
- A major rewrite of numerical experiment handling capabilities in `fe_range`. In particular uses as scheduler of hierarchical parametric computations and post-processor for parametric results are documented.
- Continued extension of file generation with capabilities to generate multiple movies `fecom AnimMovie`, save the figure in multiple formats (`.fig,.png`, see `comgui ImWrite`), ...
- extensions Java based result tables with introduction of export capabilities, see `comstr -17`.
- continued development of the `piezo` manual. In particular extension of patch meshing in `sdtweb('d_piezo#MeshPlate')` and introduction of a shunt damping example (`sdtweb('shunt')`)

Key changes for FEMLink are

- `ans2sdt` significant enhancements to `CDB` reading and writing with extended element and material/section property support. Performance fixes for `emat` reading.
- `nasread` fixed problems with multiple case importing, implementation of gravity translation. Automated writing of superelements as DMIG.
- `abaqus` significant robustness enhancements of `.INP` reading. Robustness of the `BuildUp` command.
- GUI in `femlink`

For MATLAB compatibility see section 1.8.3 .

1.8.2 Detail by function

beam1	supports the Beam1In flag to avoid use of inertia correction.
cbush	fixed problems with use of node in element for local orientation definition and hysteretic damping computations.
comgui	major extensions of the image writing capabilities. Export to Word and PowerPoint is now included with the project definition. Movie generation with the .gif extension and multi-extension (for example .png and .fig) are now included. Improved cropping options. Improved file name generation and interaction with projects, see comgui PlotWd. Introduction of style sheets, see OsDic.
fe_coor	robustness enhancements for enforced motion problems opt(1)=4.
fe_cyclic	enhanced support of long and double formats used to store solutions of cyclic and periodic systems. See also the unsupported fe_homo function.
fe_def	SubDef enhanced to support HDF formats and exports from fe_time results. SubCh and SubDOF extensions. CurveJoin supports definition of interpolation functions.
fe_exp	Major rewrite to support all expansion methods based on a reduced superelement.
fe_gmsh	Notable extension of job writing call to GMSH capabilities.
fe_mknl	Robustness of map merging.
fe_range	Major extension of GUI for visualization of experiments and the command loop for manipulating hierarchical computations.
fe_reduc	Minor robustness enhancements and introduction of a Free float=2 option where loads are orthogonalized with respect to known modes which improves conditioning.
fe_sens	Significant robustness and interactivity enhancements of GUI. Extension of fe_sens tdofTable.
fe_simul	GUI for mode computations.
fe_time	Rewrite of follow timers to allow checks during time integration.
feutil	setMat "Steel" Rho=1e-9 is how accepted. Significant extensions of set manipulation capabilities. Extended support of silent operation when using commands ending with an ;. Major rewrite of Refine operations for improved performance. Fixes to FixMpcMaster. GeomSmoothEdge places edge mid-node based on normals. Robust operation of GetDD to recover constitutive properties. Node manipulation operators (symmetry, shift, ...). performance and robustness enhancements for matching and connectivity based algorithms.

fecom	fecom AnimMovie supports generation of a directory of animated shapes. ColorMatId is documented. Synchronized figures are used in ii_mac. TextMatId places a label in the middle of each material area.
fegui	CritFcn provides a first documentation of criterion functions used to formalize objective functions with coloring and threshold strategies.
fesuper	Robustness enhancements in SE generation, stack splitting, and handling of v_handle.
id_rc	QualTable sensor sets by mode and I/O pair.
id_rm	Added new commands: PermuteIO to ease handling of hammer tests, FlipdataSign to handle sensors with flipped sign, Mass for low level implementation of modal mass estimation.
idcom	Rewrite of generalized mass estimation. Rewrite of menu structure.
ii_mac	MacCo was extended with a ByMode option. Options .MinMAC, .Df documented for pairing. Continued efforts in output formatting and export to Word/Excel/TeX. ShowDock documented to obtain MAC and two deformations, _PlotMacTick accessible from main menu Edit view, MacText accessible from context menu display text values. MacError table supports navigation in modes.
ii_mmif	robustness enhancements vel,acc, FFT properly detects spatial and time transforms. Stats introduced to handle typical feature extraction, MMIF ByIn/Out.
ii_plp	plp supports significant extensions of line markers.
iicom	Extended export to word. Notable robustness and documentation enhancements to all comgui objSet commands.
iimouse	compatibility with recent evolutions of MATLAB. Improved datatip handling. Rewrite of context menus. Support of ViewClone for multiple feplot figures. Subfunction LinkedCh was extended to enhance table interactivity with more table types (Java tables in particular). Docking commands how support topology definition.
of_mk	improved parallel operations and compatibility with MATLAB 2016a (affects multiple OpenFEM functions).
p_beam	improved support of stress computations.
p_piezo	Robustness enhancements in the support of multiple consitutive law formats.
p_shell	formulations 5x support rotation inertia for explicit time integration. SetThick can be used to introduced variable thickness areas.
p_zt	implementation of a zero thickness element family used to constrain normal and tangential displacement on possibly curved interfaces.

sdtcheck	robustness enhancements in particular for PatchJavapath which is necessary for new GUI operations.
sdtroot	has been notably extended with many new tabs, support floating tabs in GUI development, ... Related changes affect sdt_locale, sdcedit, sdt_dialogs, ...
sdtweb	notable extensions of support commands _wd, _find, ...

1.8.3 Notes by MATLAB release

- MATLAB 7.14 (2012a) to 9.0 (2016a). *SDT & FEMLink 6.8* are developed for these versions of MATLAB and are fully compatible with them.
- MATLAB 7.6 (2008a) to 7.13 (2011b). *SDT & FEMLink 6.8* are being phased out but can be used for a number of operations.
- On Linux MATLAB 8.1 (2013a) is required.
- Earlier MATLAB releases are no longer supported.
- MATLAB 8.5 has known bugs in the handling of colorbar.
- MATLAB 7.9 has known compatibility problems in its HDF library and should be avoided for large FEM applications using sdthdf.

1.9 Release notes for SDT and FEMLink 6.7

1.9.1 Key features

SDT 6.7 is the only version compatible with MATLAB 8.4 (2014b) and later. Key changes of this release are

- A major revision of all the SDT interfaces for compatibility with the new graphical system of MATLAB 8.4 (2014b).
- Major extensions legend/filename generation capabilities, see `comgui`.
- Significant enhancement of documentation with more readable links to be reused in `sdtweb` calls.
- introduction of more readable Java based results tables in many functions.

Key changes for FEMLink are

- `ans2sdt` major extension of CDB reading capabilities and bug fixes associated to changes in MATLAB R2013a behavior and 64 bit pointers in newer ANSYS versions. Reading of mapping is now supported. Reading of stresses and other ESL output in `.rst` files.
- `nasread` better handling of CROD cases that correspond to `bar1`. Support of rectangular DMIG writing. Enhanced `PBEAM,PROD` translation.
- `abaqus` more consistent reading of steps the `BuildCase` command allows setting-up the case relative to a desired step. Revision of the `resolve` command to enhance handling of node and element sets in general cases (compatibility with assembly of part instances). Translation of contact (for the `*CONTACT PAIR` command) to the contact module or SDT/NL, see section 1.4 , is now supported for most classical rules. `*Orientation` properly translated into `pro.MAP`.
- `samcef` improved multiple files with `INPUT` cards. Detailed improvement of `.AEL, .FRA, .MCT, .STI, ...`

For MATLAB compatibility see section 1.9.3 .

1.9.2 Detail by function

comgui	major revision of image (ImWrite), filename (ImFtitle) and legend (def.Legend) generation utilities. Major extensions of the grouped object setting objSet commands used by SDT for figure and object formatting. Name generation extended and properly documented in objString Export of java tables to LaTeX/csv/text formats.
cinguj	major rewrite of the JAVA based GUI part of the toolbox with many bug fixes. A static javapath is now desired and fixed with sdtcheck('PatchJavaPath').
fe2ss	introduced -cpx command option for complex mode state-space support where hysteretic damping models can be used. This complements the alternative -loss2C. fe2ss SysDef now support restitution of enforced displacement.
feplot	significant extensions of colormaps. Full rewrite of fecom AnimMovie.
feutil	extended support for non OpenFEM face numbering schemes. OptimDegen transforms degenerate elements to their lower node number counterpart. Extended support of silent mode with ; at end of command.
feutilb	provides commands CombineModel and SubModel to support combination of models or extraction of submodels with clean handling of Stack, Mat/Pro, Case entries.
fe_case	more robust handling of hysteretic damping assembly for interpolated materials. Introduced fe_case SetCurve commands to easier handling of time/frequency varying loads.
fe_ceig	first order correction is now supported with CeigMethod=2.
fe_cyclic	support of inertial loading on partial model.
fe_eig	GenMass command generates table of generalized masses.
fe_exp	major rewrite of function and documentation to support newer data structures and optimize performance.
fe_load	improved support of DofLoad.
fe_mat	robustness of unit handling, extensions of material law interpolation.
fe_mpc	FixDofBas2mpc supports transform of local basis FixDof to MPC entries. DofSetMerge combines multiple DOFSet into one, which is the only case supported by most solvers.
fe_sens	tdofTable handling of sensor definitions as tables has been further documented and robustified.

fe_range	significantly extend commands previously in fe_def('range') for DOE handling.
fe_simul	extended support of damping and enforced input (DofSet) entries in direct frequency response.
fe_time	support for enforced displacement with DOFSet entries has been introduced.
fe_time	significant extensions and performance enhancements for explicit solvers.
fe_mknl	optimized support for node numbers above 2e9. Robustness and documentation extensions for pro.MAP.
fe_stress	computation of Von Mises stress is now compiled.
iicom	significant extension of support for 2D plots (contour, image, ...). Robustness and documentation improvements of multiple image generation ImWrite.
iimouse	major rewrite for R2014b compatibility, interactivity with java tables (comstr -17). Support of docking and datatip interactivity. Additional keyboard callbacks (press ? in iiplot or feplot).
id_rc	enhanced support of error and quality indicators of identification quality.
id_rm	robustness and GUI enhancements.
ii_mac	significant GUI rewrite in particular for table generation.
ii_mmif	robustness enhancements for signal processing capabilities.
ii_plp	major extensions of Legend, TickFcn, introduction of new ColorMap
m_piezo	support for a database of commercial piezo patches has been introduced. Materials have been added and properties corrected.
nor2ss	fixed compatibility issues with the Control Toolbox and Simulink. Improved warnings.
p_piezo	introduced new Tab commands to display model and material properties in a easily readable forms.
p_solid	improved support of anisotropic materials and composite shell.
res2ss	notable rewrite and fixed compatibility issues with the Control Toolbox and Simulink.
sdtweb	significant robustness enhancements. Improved _TagList. Introduction of _find command for file search.
setlines	improved compatibility with comgui objSet.
ufread,ufwrite	improved compatibility LMS-Testlab and performance enhancements.
cbush,celas	improved support for gyroscopic matrices and loss factor.

1.9.3 Notes by MATLAB release

- MATLAB 7.6 (2008a) to 8.5 (2015a). *SDT & FEMLink 6.7* are developed for these versions of MATLAB and are fully compatible with them.
- MATLAB 7.9 has known compatibility problems in its HDF library and should be avoided for large FEM applications using sdthdf.

- Earlier MATLAB releases are no longer supported.

Modal test tutorial

2.1 iiplot figure tutorial 42
2.1.1 The main figure 43
2.1.2 The curve stack 46
2.1.3 Handling what you display, axes and channel tabs 48
2.1.4 Channel tab usage 49
2.1.5 Handling displayed units and labels 50
2.1.6 SDT 5 compatibility 50
2.1.7 iiplot for signal processing 52
2.1.8 iiplot FAQ 53
2.2 Identification of modal properties (Id dock) 54
2.2.1 Opening and description of used data 54
2.2.2 General process 59
2.2.3 Importing FRF data 63
2.2.4 Write a script to build a transfer structure 65
2.2.5 Data acquisition 66
2.3 Pole initialization (IdAlt and IdMain filling) 67
2.3.1 External pole estimation 67
2.3.2 LSCF 68
2.3.3 Single pole estimate 74
2.3.4 Band to pole estimate 76
2.3.5 Direct system parameter identification algorithm 76
2.3.6 Orthogonal polynomial identification algorithm 77
2.4 Identification options 77
2.5 Estimate shapes from poles 79
2.5.1 Broadband, narrowband, ... selecting the strategy 79
2.5.2 `Qual`: Estimation of pole and shape quality 82
2.5.3 When `id_rc` fails 88

2.6 Update poles 91
2.6.1 Eup : for a clean measurement with multiple poles 91
2.6.2 Eopt : for a band with few poles 92
2.6.3 EupSeq and EoptSeq : sequential narrowband pole updating . . 93
2.6.4 Example for practice 93
2.6.5 Background theory 96
2.7 Display shapes : geometry declaration, pre-test 97
2.7.1 Modal test geometry declaration 97
2.7.2 Sensor/shaker configurations 99
2.7.3 Animating test data, operational deflection shapes 101
2.8 MIMO, Reciprocity, State-space, ... 103
2.8.1 Multiplicity (minimal state-space model) 103
2.8.2 Reciprocal models of structures 105
2.8.3 Normal mode form 107

An experimental modal analysis project can be decomposed in following steps

- before the test, preparation and design (see section 2.7)
- after data acquisition, import into the SDT (see section 2.2)
- navigation through data in the `iiplot` figure (see section 2.1)
- identification procedure :
  - initialize the pole list (see section 2.3)
  - setup the identification options (see section 2.4)
  - identify the pole residues and evaluate the identification quality (see section 2.5)
  - optimize poles to improve the identification quality (see section 2.6)
- handling of MIMO tests and other model transformations (output of identified models to state-space, normal mode, ... formats, taking reciprocity into account, ...) (see section 2.8)

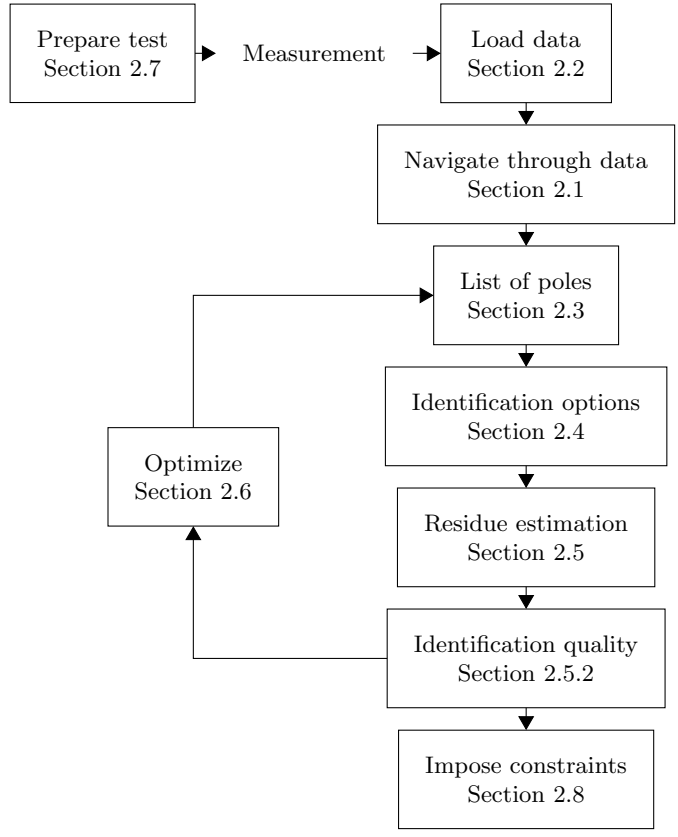


Figure 2.1: Modal test protocol with links to corresponding sections

Further steps (test/analysis correlation, shape expansion, structural dynamics modification) are discussed in chapter section 3 .

2.1 iiplot figure tutorial

`iiplot` is the response viewer used by SDT. It is essential for the identification procedures but can also be used to visualize FEM simulation results.

As detailed in section 2.2 , 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.

2.1.1 The main figure

For simple data viewing you can open an `iiplot` figure using `ci=iiplot` (or `ci=iiplot(2)` to specify a figure number). For identification routines you should use `ci=idcom` (standard datasets are then used see section 2.2).

To familiarize yourself with the `iiplot` interface, run `demosdt('demogartidpro')`. Which opens the `iiplot` figure and the associated `iiplot(2) properties` figure whose tabs are detailed in the following sections.

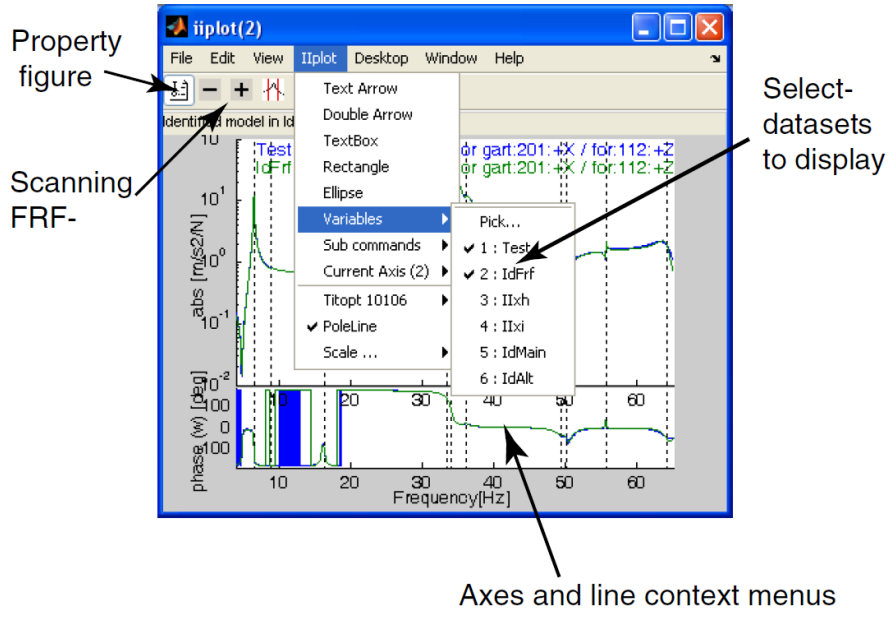


Figure 2.2: Display figure of the `iiplot` interface.

Toolbar

Toggles the display or not of the iiplot property figure.

Previous channel/deformation, see iicom ch+.

Next channel/deformation.

Fixed zoom on FRF, see iicom wmin. Note that the variable zoom (drag box) is always active, see iimouse zoom.

Start cursor, see iimouse Cursor.

Refresh the displayed axes.

No subplot. See iicom Sub[1,1].

2 subplots. See iicom Sub[2,1].

Amplitude and phase subplots. See iicom Submagpha.

switch lin/log scale for x axis. See iicom xlin.

switch lin/log scale for y axis. See iicom ylog.

switch lin/log scale for z axis. See iicom xlog.

Show absolute value. See iicom Showabs.

Show phase. See iicom Showpha.

Show real part. See iicom Showrea.

Show imaginary part. See iicom Showima.

Show real and imaginary part. See iicom Showr&i.

Show Nyquist diagram. See iicom Shownyq.

Show unwrapped phase. See iicom Showphu.

Snapshot. See iicom ImWrite

Mouse operation and keyboard shortcuts

Mouse and keypress operations are handled by iimouse within iiplot, feplot, and ii_mac figures. For a list of active keys press ? in the current figure.

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 need to type the i key with the axis of interest active. An axis becomes active when you click on it.

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. Note how the left click gives you detailed information on the current point or the left click history. In iiplot you can for example use that to measure distances.

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.

Context menus

The axes ContextMenu (click on the axis using the right mouse button) lets you select , set axes title options, set pole line defaults, ...

- Cursor tracks mouse movements and displays information about pointed object. For ODS cursor see iicom ods.
- Show chooses what to display.
- Compute... [MMIF,CMIF...] chooses what to compute and display. The iicom('show [MMIF,CMIF...]') command line is similar. Details on what can be computed are given in ii_mmif.
- Variables in current axis... chooses which variable to display, see iicom IIx.
- iiplot properties, same as iicom('pro'), opens the property figure.
- Scale...[x lin, x log...] chooses the axis scale as the. See iicom xlin or use iimouse('axisscale[xlin,xlog...]') commands.
- TitOpt chooses the title, axis and legend labels-format.
- PoleLine pole line selection.
- Views... chooses the views, see iimouse view.
- colorbar shows the colorbar and is equivalent to cingui('ColorBarMenu') command line.
- Zoom reset is the same as the iimouse('resetvie') command line to reset the zoom.
- setlines calls the associated function.

The line ContextMenu lets you can set line type, width, color ...

The title/label ContextMenu lets you move, delete, edit ... the text

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).

2.1.2 The curve stack

iiplot considers data sets in the following format

- Response data related to UFF58 format
- Curves generated by SDT
- Shapes at DOFs related to UFF55 format

This data is stored in iiplot figures as a Stack field (a cell array with the first column giving 'curve' type entries, the second giving a name for each dataset and the last containing the data, see stack_get). To allow easier access to the data, *SDT* handle objects are used. Thus the following calls are equivalent ways to get access to the data

```
ci=iicom('curveload','gartid');
iicom(ci,'pro');iicom(ci,'CurTab Stack'); % show stack tab

% Normal use : the figure pointer stack
ci.Stack % show content of iiplot stack
ci.Stack{'Test'}  % a copy of the same data, selected by name
ci.Stack{1,3}     % the same by index
% Use regular expresion ('II.*' here) for multiple match
ci=stack_rm(ci,'curve','#II.*')

% If you really insist on low level calls
gf=sdtdef('cf'); % recover current sdth handle, number may vary
r1=get(gf,'userdata'); % object containing the data (same as ci)
s=ci.vfields.Stack.GetData % get a copy of the stack (cell array with
           % type,name,data where data is stored)
s{1,3}     % the first data set

%  Alternative use (obsolete) : the XF stack pointer
```

```
XF1=iicom(ci,'curvexf');
XF1('Test')        % still the same dataset, indexed by name
XF2=XF1.GetData;   % Copy the data from the figure to variable XF2
```

The `ci.Stack` handler allows regular expression based access, as for `cf.Stack`. The text then begins by the `#` character.

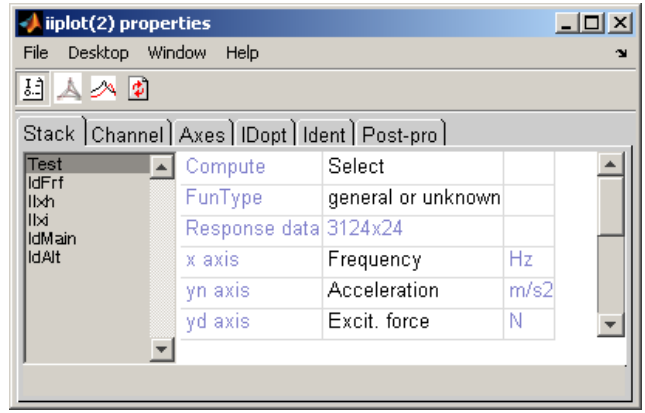


Figure 2.3: Stack tab of the `iiplot` interface.

The graphical representation of the stack shown in figure 2.3 lets you do a number of manipulations witch are available trough the context menu of the list of datasets in the stack

- `Compute` gives access to data processing commands in `ii_mmif`. You perform the analysis from the command line with `iicom(ci,'sum','Test')`. The list of available post processing functions is given by `ii_mmif list`.
- `Load` lets you load more data with `iicom(ci,'curveload-append','gartid')`, replace the current data with `iicom(ci,'curveload','gartid')`
- `Display` lets you display one or more selected dataset in the iiplot figure (see corresponding command `iicom IIx`).
- `Save` lets you save one or more dataset (see corresponding command `iicom CurveSave`).
- `Join` combines selected datasets that have comparable dimensions (see corresponding command `iicom CurveJoin`).
- `Cat` concatenates selected datasets along time or frequency dimension (see corresponding command `iicom CurveCat`).

- `Remove` removes selected dataset (see corresponding command `iicom CurveRemove`).
- `NewId` opens a new `idcom` figure with the selected dataset (see corresponding command `iicom CurveNewId`).

2.1.3 Handling what you display, axes and channel tabs

`iiplot` lets you display multiple axes see `iicom Sub`. Information about each axis is show in the `axes` tab.

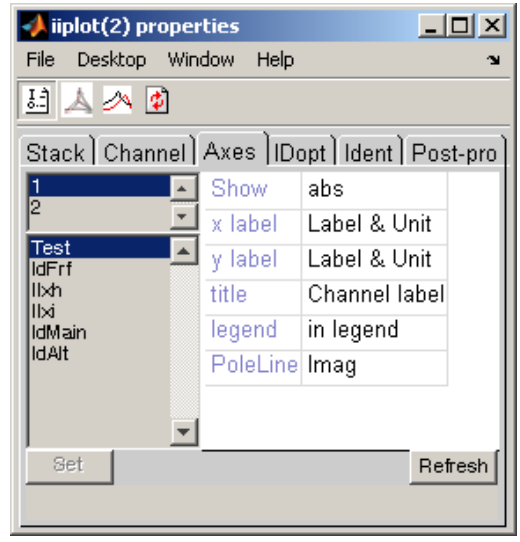


Figure 2.4: Axes tabs of the `iiplot` interface.

For example open the interface with the commands below and see a few thing you can do

```
ci=idcom;iicom(ci,'CurveLoad sdt_id');
ci.Stack{'curve','IdFrf'}=ci.Stack{'Test'}; % copy dataset
ci.Stack{'IdFrf'}.xf=ci.Stack{'Test'}.xf*2; % double amplitude
iicom('CurTab Axes');
```

- `Sub` Subplots : 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 using `ci=iiplot(2); iicom(ci,'SubMagPha')` gives you control on which figure the command applies to.

- `Show` 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 selecting `phase(w)` in the **axis type menu** (which is located just on the right of the current axis button).
- `IIx` select sets you want to display using `iicom(';showabs;ch1');` `iicom('iix only',{'Test','IdFrf'})`. You could also achieve the same thing using the `IIplot:Variables` menu.
- Note that when you print the figure, you may want to use the `comgui('ImWrite','FileName.ext')` command or `-noui` switch so that the GUI is not printed. It is the same command as for feplot image printing (see `iicom ImWrite`).

2.1.4 Channel tab usage

Once you have selected the datasets to be displayed, you can use the channel tab to scan trough the data. Major commands you might want to know

- 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 `Channel` tab of the property figure (open with `iicom('InitChannel')`) and select one more than one channel in the list. In the figure, the `>10` is used to illustrate that the tab supports channel selection. For datasets with string labels use `10*`.
- Note that you can also select channels from the command line using `iicom('ch 1 5')`.

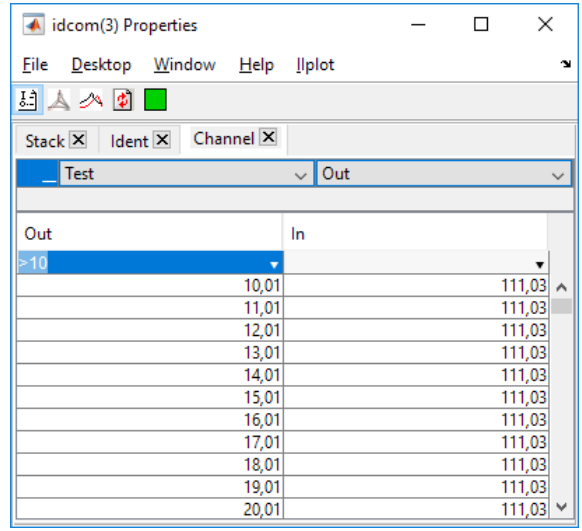


Figure 2.5: Channel tabs of the `iiplot` interface.

2.1.5 Handling displayed units and labels

```
ci=iicom('curveload gartid');
ci.Stack{'Test'}.yn.unit='N';
ci.Stack{'Test'}.yd.unit='M';
iicom sub
```

2.1.6 SDT 5 compatibility

With *SDT 6*, global variables are no longer used and `iiplot` supports display of curves in other settings than identification.

If you have saved *SDT 5* datasets into a `.mat` file, `iicom('CurveLoad FileName')` will place the data into an *SDT 6* stack properly. Otherwise for an operation similar to that of *SDT 5*, where you use `XF(1).xf` rather than the new `ci.Stack{'Test'}.xf`, you should start `iiplot` in its identification mode and obtain a pointer `XF` (*SDT* `handle` object) to the data sets (now stored in the figure itself) as follows

```
>> ci=iicom('curveid');XF=iicom(ci,'curveXF')
```

```
XF (curve stack in figure 2) =

XF(1) : [.w    0x0, xf    0x0] 'Test'  : response (general or unknown)
XF(2) : [.w    0x0, xf    0x0] 'IdFrf'  : response (general or unknown)
XF(3) : [.w    0x0, xf    0x0] 'IIxh'  : response (general or unknown)
XF(4) : [.w    0x0, xf    0x0] 'IIxi'  : response (general or unknown)
XF(5) : [.po   0x0, res   0x0] 'IdMain'  : shape data
XF(6) : [.po   0x0, res   0x0] 'IdAlt'  : shape data
```

The following table lists the global variables that were used in *SDT 5* and the new procedure to access those fields which should be defined directly.

`XFdof`	described DOFs at which the responses/shapes are defined, see `.dof` field for response and shape data in the `xfopt` section, was a global variable pointed at by the `ci.Stack{'name'}.dof` fields.
`IDopt`	which contains options used by identification routines, see `idopt`) is now stored in `ci.IDopt`.
`IIw`	was a global variable pointed at by the `ci.Stack{'name'}.w` fields.
`IIxf`	(main data set) was a global variable pointed at by the `ci.Stack{'Test'}.xf` fields.
`IIxe`	(identified model) was a global variable pointed at by the `ci.Stack{'IdFrf'}.xf` fields.
`IIxh`	(alternate data set) was a global variable pointed at by the `ci.Stack{'IIxh'}.xf` fields.
`IIxi`	(alternate data set) was a global variable pointed at by the `ci.Stack{'IIxi'}.xf` fields.
`IIpo`	(main pole set) was a global variable pointed at by the `ci.Stack{'IdMain'}.po` fields.
`IIres`	(main residue set) was a global variable pointed at by the `ci.Stack{'IdMain'}.res` fields.

`IIpo1` (alternate pole set) was a global variable pointed at by the `ci.Stack{'IdAlt'}.po` fields.

`IIres1` (alternate residue set) was a global variable pointed at by the `ci.Stack{'IdAlt'}.res` fields.

`XF` was a global variable pointed holding pointers to data sets (it was called a database wrapper). The local pointer variable `XF` associated with a given `iiplot` figure can be found using `CurrentFig=2;ci=iiplot(CurrentFig); XF=iicom(ci,'curveXF')`.
The normalized datasets for use with `idcom` are generated using `ci=idcom;XF=iicom(ci,'curvexf')`. They contain four response datasets (`XF('Test')` to `XF('IdFrf')`) and two shape datasets (`XF('IdMain')` and `XF('IdAlt')`).

2.1.7 iiplot for signal processing

`iiplot` figure lets you perform standard signal processing operations (FFT, MMIF, filtering...) directly from the GUI. Opening `iiplot` properties figure, they are accessible trough the contextual menu `compute` (right click on the curve list in the Stack tab). Once an operation has been performed, its parameters can be edited in the GUI, and it can be recomputed using the `Recompute` button.

Following example illustrates some signal processing commands.

```
[mdl,def]=fe_time('demobar10-run'); % build mdl and perform time computation
cf=feplot(2); cf.model=mdl; cf.def=def;

ci=iiplot(3);
fecom(cf,'CursorOnIiplot') % display deformations in iiplot

% all following operations can be performed directly in the GUI:
% see the list of curves contained in iiplot figure, Stack tab:
iicom(ci,'pro');iicom(ci,'curtab Stack');
% compute FFT of deformations.  Name of entry 'feplot(2)_def(1)'
ename=ci.Stack(:,2); ename=ename{strncmp(ename,'feplot',5)};
ii_mmif('FFT',ci,ename) % compute
fname=sprintf('fft(%s)',ename);
iicom(ci,'curtab Stack',fname); % show FFT options that are editable
  % edit options & Recompute:
ci.Stack{fname}.Set={'fmax',50};
iicom(ci,'curtab Stack',fname,'Recompute');
```

```
% filter and display (the bandpass removes a lot of transient)
ii_mmif('BandPass -fmin 40 -fmax 50',ci,ename) % compute
fname=sprintf('bandpass(%s)',ename);
ci.Stack{fname}.Set={'fmin',10,'fmax',20};
iicom(ci,'curtab Stack',fname,'Recompute');
iicom(ci,'iix',{ename,fname});
```

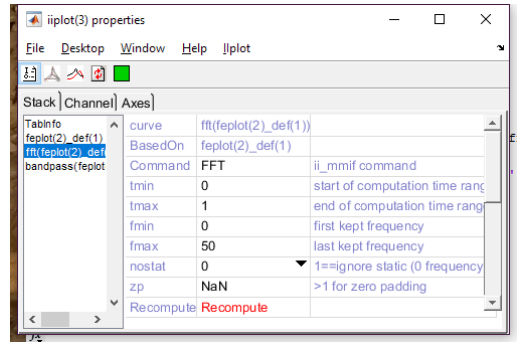


Figure 2.6: GUI for FFT computation

2.1.8 iiplot FAQ

This section lists various questions that were not answered elsewhere.

- **How do I display a channel with an other channel in abscissa?**
  The low level call `ci.ua.ob(1,11)=channel;` defines the channel number `channel` of the displayed curve as the abscissa of other channels.

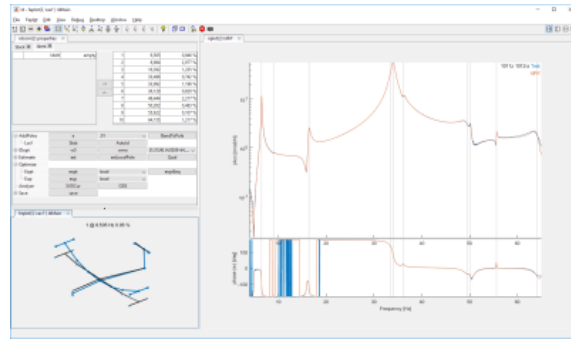
  ```
  ci.ua.ob(1,11)=3; % define channel 3 as abscissa
  iiplot;           % display the changes
  set(ci.ga,'XLim',[0 1e-3]); % redefine axis bounds
  ```

- **Channel selection in multi-dimensional arrays**

  ```
  %  sdtweb('demosdt.m#DemoGartteCurve') % FRF with 2 damping levels
  ci=iiplot(demosdt('demogarttecurve'))
  ci.Stack{'New'}
  iicom(ci,'ChAllzeta')
  ```

2.2 Identification of modal properties (Id dock)

Identification is the process of estimating a parametric model (poles and modeshapes) that accurately represents measured data. The identification process is typically performed using the `dock` shown below opened with `iicom('dockId')`.



2.2.1 Opening and description of used data

The following procedure loads data from a .unv file but other way to open and load data are available.

- Open an empty dock `iicom('dockid')` and load data from the interface by selecting files (see below). A list of acquisition software from which data have been successfully loaded is described in section 2.2.3 .
- Reopen a dock previously saved in SDT format (`.mat`).
  - For saving : in `idcom` figure, use File:Save, chose the data that need to be saved (all selected by default) and then chose the saving file name.
  - For reloading: execute the command `iicom('curveLoad File.mat')`
- Load data from variables in the workspace. It is then possible to load data from files directly into variables (see section 2.2.3 , which is useful if data customization is required) or to deal with user-built transfers (see section section 2.2.4) and finally pass the result to `Id` dock.

```
% Unv with wire-frame, transfer and poles
% Open empty dockid get pointer to feplot (cf) and iiplot (ci)
[ci,cf]=iicom('dockid');
% Build gartid.unv file the first time, then provide file name
fname=demosdt('build gartid.unv');
```

```
% Data are stored into a variable to help you build custom loading procedure
UFS=ufread(fname);
wire=UFS(1); % Test wireframe
XF=UFS(2); % Transfers
ID=UFS(3); % List of modes
cf.mdl=wire; % Store the wireframe in the feplot figure
% Put transfers to iiplot figure (Transfers named test are the ones
ci.Stack{'curve','Test'}=XF; concerned by the current identification)
ci.Stack{'curve','IdMain'}=ID; % Store the poles in the iiplot figure
iicom('iix:TestOnly'); % Equivalent to :  idcom figure, tab Stack,
% right click on Test and select 'Display selected data'
```

When manual assignation is performed, do not forget to click on to refresh the tables (for instance the pole list in `idcom`). Note that to perform identification, only the transfers are needed: the wireframe allows visualizing the identified mode shapes and the list of poles is helpful if previous identification has been performed.

On top of the `Test` and `IdMain` data discussed above, other useful data used throughout the identification process and stored in the `iiplot` Stack are

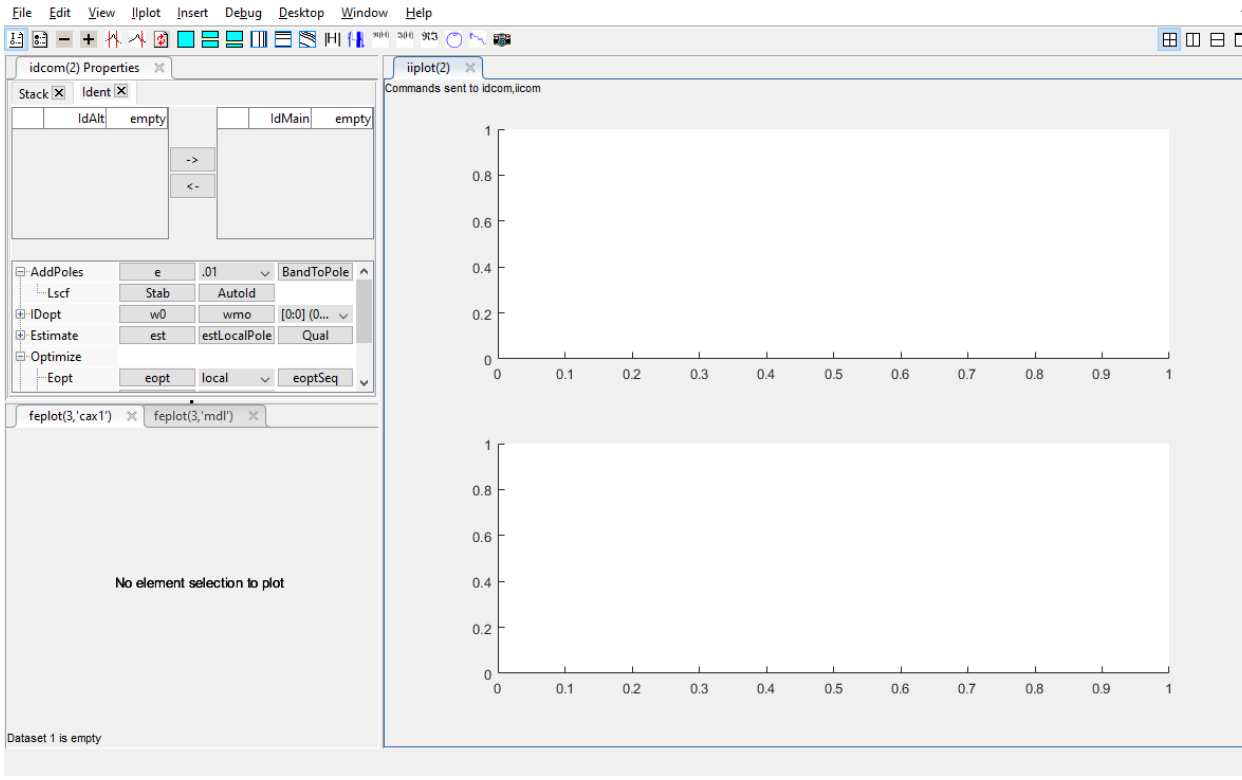
- `Test` contains measured frequency response functions. See section 2.2.3 ways to initialize this data set.
- `IdFrf` contains the synthesis of transfers associated with given set of transfers (shown in red in the figure above).
- `IdAlt` contains the alternate set of modes (poles and residues). These are listed on the left list of the `Ident` tab below.
- `IdMain` contains the main set of modes (poles and residues). These are listed on the right list of the `Ident` tab.

```
[ci,cf]=gartid; % Open dockid with stored data and performs identification
ci.Stack % Display list of stored data in the Stack of iiplot

Test=ci.Stack{'curve','Test'}; % Retrieve data from iiplot
IdFrf=ci.Stack{'curve','IdFrf'};
IdMain=ci.Stack{'curve','IdMain'};
IdAlt=ci.Stack{'curve','IdAlt'};

wire=cf.mdl.GetData; % GetData is used to retrieve a copy.
% Otherwise all modifications are propagated to feplot
```

Here is a tutorial for interactive data loading in `DockId`

You will need the `garteur` example files, which can be found in *SDTPath/sdtdemos/gart*.m*. If these files are not present, click on the first step on the following tutorial in the HTML version of the documentation or download the patch at the adress https://www.sdtools.com/contrib/garteur.zip and unzip the content in the the folder *SDTPath/sdtdemos*.

1. Execute the command `iicom('dockid')` to open an empty dock.

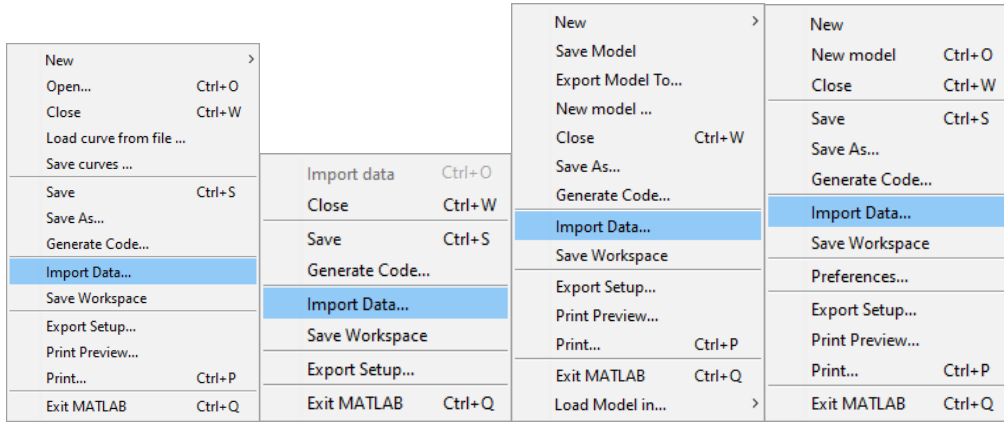


The dock is divided in three parts:

- At right, the `iiplot` figure where are displayed all curves (measured transfers, synthesized transfers, mode indicators...)
- At the top left hand corner, the `idcom` figure which is used to interact with the data in `iiplot`, especially here using the `Ident` tab to perform the identification process
- At the bottom left hand corner, the `feplot` figure where the wireframe is displayed. It lets you animate the identified modeshapes. The `feplot('mdl')` is accessible behind and lets you visualize the information about the wireframe.

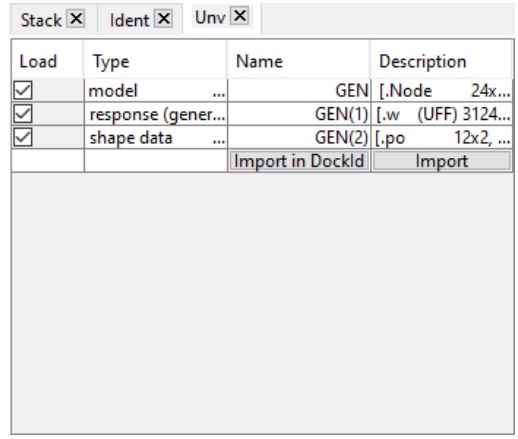
2. ▶ The loading of `.unv` files can be realized from `iiplot` or `feplot`. Activate for instance the `idcom` figure and select `File:ImportData...`

   Here are the 4 possible menus in this order: `iiplot`, `idcom`, `feplot` and `feplot('mdl')`.



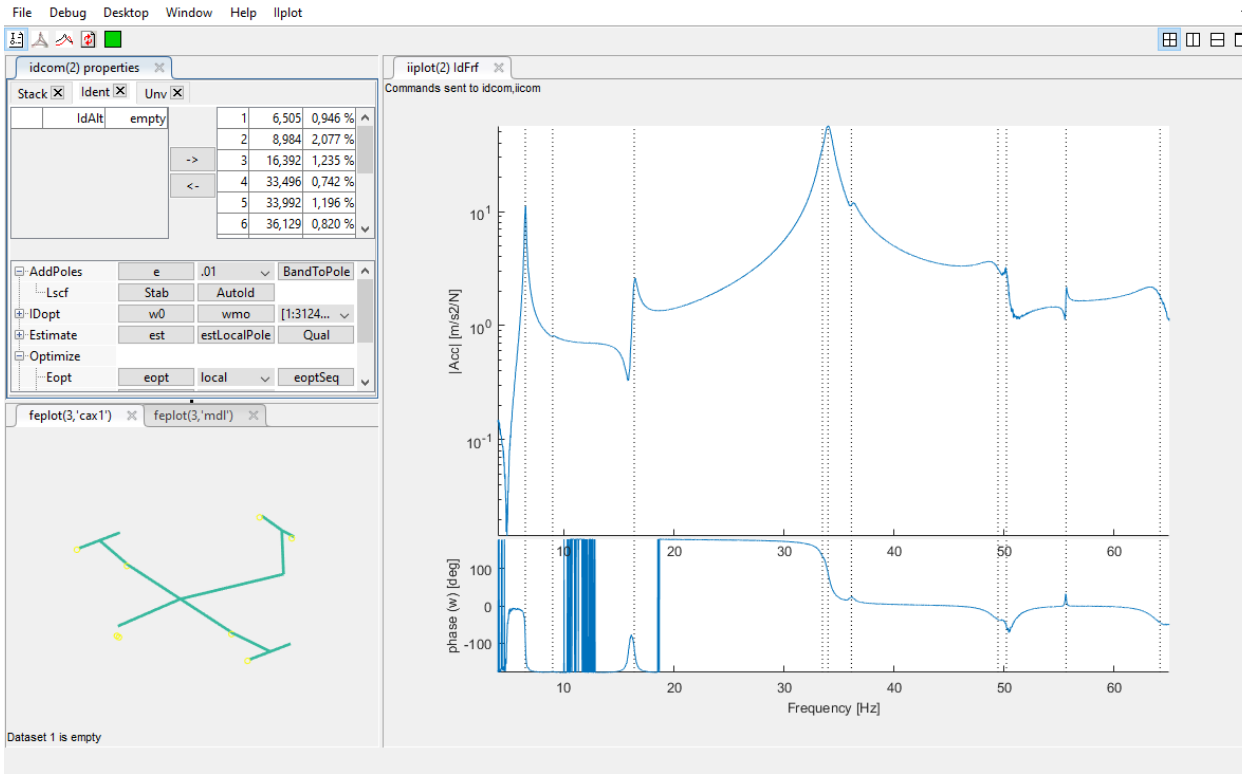
   In the opening window, select the file to load. For this tutorial, the file is located at *SDT-Path/sdtdemos/gartid.unv*.

   Once selected, the `Unv` tab is displayed in the `idcom` or the `feplot('mdl')` figure (depending the chosen menu for `ImportData`.



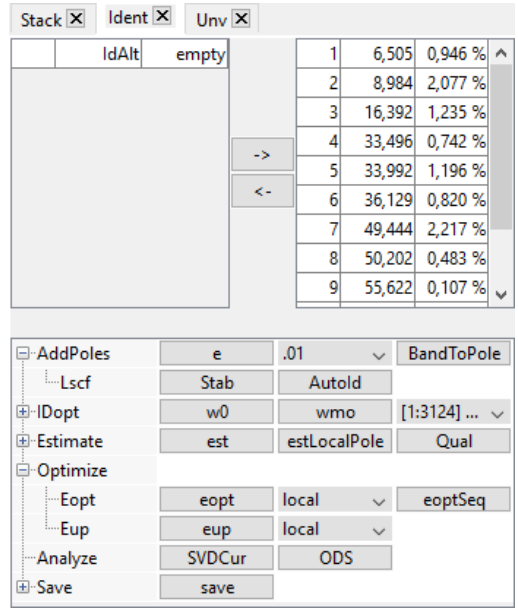
   It shows that three types of data are present in the file: a wireframe, transfers and identified mode shapes. Select the three check boxes to load everything.

3. ▶ Click on `Import` (or `Import in DockId` which is used to build dockId if the loading is performed in a `feplot` or an `iiplot` figure outside a dockid).



The data are loaded: transfers are shown in the `iiplot` figure, the wireframe in the `feplot` figure and the list of poles in the tab `Ident` of the `idcom` figure.

4. ▶ Once an identification is performed, click on `Save` in the `idcom` figure.



A windows pops-up to ask what data must be saved. Save all (by default) to set all the data and info on the dockid in the saving file.

Close the dock. A pop-up should appear to ask if you really want to close `iiplot` (this is to ensure that no data is lost if no saving has been performed), click on `Close without saving`.

5. To reload the saved `dock`, two possibilities are available:
   - Execute the command `iicom('curveload filename')`
   - Open an empty iiplot figure and load the saved file with `File:Import Data...`

2.2.2 General process

The proposed identification process is outlined below. The main steps of the methodology are

- Initial pole estimates are placed in `IdAlt` using advanced pole picking, LSCF (see section 2.3) or any other algorithm outside SDT.
- A user validated list of poles is kept in `IdMain`. The arrows between the two list in the interface (which correspond to the `ea` and `er` commands) can be used to move poles between the two lists: add missed poles, remove computational or undesired poles .

- Shapes (pole/residue models, residual terms, modeshapes derived from residues) are then estimated for each pole given in `IdMain`. Several strategies exist and are more deeply explained at section 2.5
  - Broad band estimation on the whole frequency band : `est` command/button
  - Narrow band estimation on the selected band : `estlocal` command/button
  - Iterative local estimation around each pole : `estlocalpole` command/button
- Optimizing poles (and residues) of the current model depending on the quality obtained by the previous passes. As for the estimation of shapes, there three strategies for the optimization:
  - Broad band update : `eup` for high number of poles and `eopt` for up to 2-3 poles
  - Narrow band update on the selected band: `euplocal` and `eoptlocal`
  - Iterative local updates around each pole: `eoptSeq`

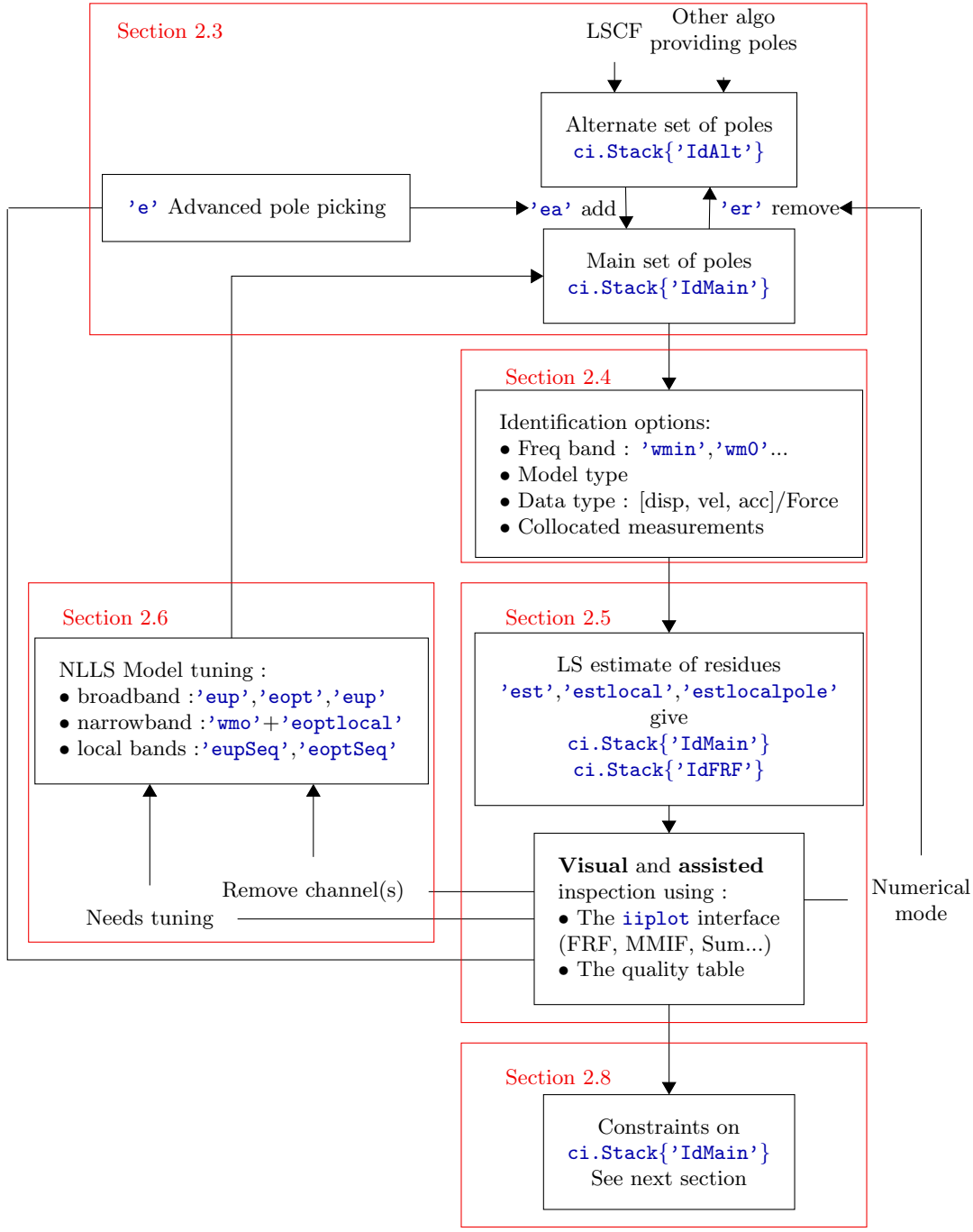
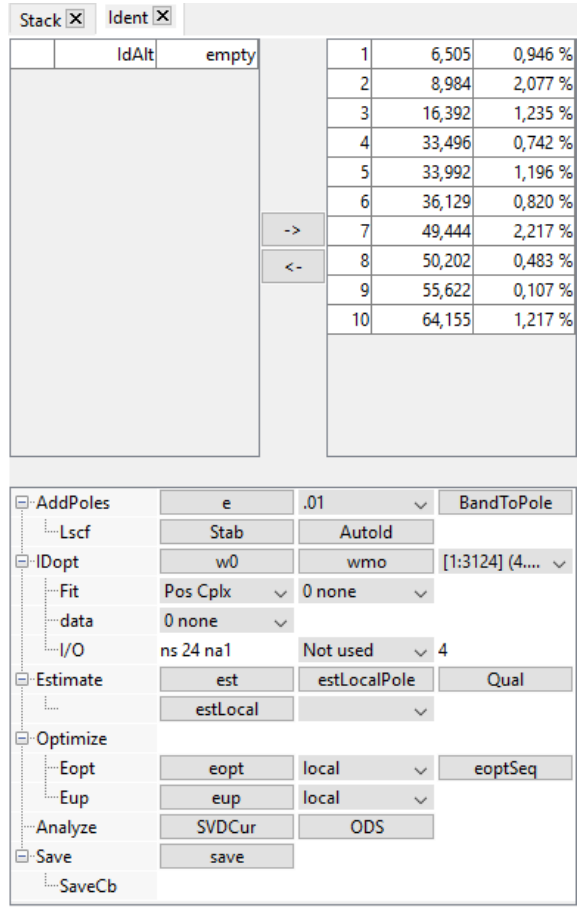


Figure 2.7: Modal identification process with links to corresponding sections

This process is handled through the `Ident` tab opened with `iicom('InitIdent')` or with the interface by clicking on `Tab` : `Ident` from the `iiplot` or `idcom` figure.



The main steps, associated with level 1 lines in the GUI tree are the topics of specific sections of the documentation:

- AddPoles : use an initial algorithm to estimate poles (single pole estimator or selection in a stabilization diagram LSCF).
- IDopt : select frequency range and possibly define properties of transfers (displacement, velocity, acceleration, MIMO,...)
- Estimate shapes using a frequency domain output error method that builds a model in the pole residue form (see section 5.6). Theoretical details about the underlying algorithm are given in section 2.6.5 . Section 2.5.3 addresses its typical shortcomings.

- Adjust poles using one of the non-linear optimization algorithms.
- Transform the output to a format dealing with MIMO constraints, reciprocity, ...

The `gartid` script gives real data and an identification result for the GARTEUR example. The `demo_id` script analyses a simple identification example.

2.2.3 Importing FRF data

SDT stores transfer functions in the `Response data` (`.w,.xf` fields) or curve (`.X,.Y` fields) formats. 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.
LMS	Export data from LMS CADA-X to UFF or MATLAB format.
Polytec	Install the `Polytec File Access` library on your computer and use the `polytec` function to import `.svd` files directly. Alternatively, export data from PSV software to UFF.
Dactron	Export data from RT-Pro software to the UFF. Use the *Active-X API* to drive the Photon from MATLAB see photon.
MathWorks	Use *Data Acquisition* and *Signal Processing* toolboxes to estimate FRFs and create a script to fill in *SDT* information (see section 2.2.3).
MTS	Export data from IDEAS-Pro software to UFF.
Spectral Dynamics	Create a Matlab script to format data from SigLab to *SDT* format.

- *Universal files* are easiest if generated by your acquisition system. Writing 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

```
% generate gartid.unv (or retrieve file name if already generated)
 fname=demosdt('build gartid.unv');
 UFS=ufread(fname);  % read the unv file
 UFS % This command display in the command window the content of the file
 xf=UFS(2); % Read the transfers in the file and store in the variable xf
```

```
%% Do everything needed with the data for customization if needed %%%
% For instance extract channels 1:4
xf=fe_def('SubDofInd',xf,1:4)
% Then pass to iiplot for view and ID purposes
ci=idcom;   % For identification purposes open IDCOM
% Store transfers in 'Test' which are transfers to be identified
ci.Stack{'curve','Test'}=xf;

% To only view data in figure(11) the following would be sufficient
cj=iiplot(11);      % open an iiplot in figure 11
iiplot(cj,UFS(1));  % show UFS(1) there
```

where you read the database wrapper `UFS` (see `xfopt`), initialize the `idcom` figure, assign dataset 2 of `UFS` to dataset 'Test' 1 of `ci` (assuming that dataset two 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 supported by `ufread` with a stared file name
`UFS=ufread('FileRoot*.unv');`

- *Polytec files* need many options to extract data (Time/Transfers, Estimator H1/H2, Velocity/Force...). Please read the dedicated `polytec` documentation to adapt the example below to your needs. Note that the code below needs `Polytec File Access` to be installed.

```
fname=sdtcheck('patchget',struct('fname','PolytecMeas.svd'));
% Provide a cell array with all readable measured data
list=polytec('ReadList',fname);
display(list);
% Extract the transfer function Vib/Ref1
% with the estimator H1 Displacement/Voltage
RO=struct('pointdomain','FFT','channel','Vib & Ref1',...
'signal','H1 Displacement / Voltage');
XF=polytec('ReadSignal',fname,RO);
% alternative call using one row of the cell array "list"
XF=polytec('ReadSignal',fname,struct('list',{list(20,:)}));
```

To avoid the manual filling of the reading options, it is also possible to simply load data from the interface : follow the tutorial in section section 2.2.1) but select the .svd file instead of the .unv file and do right-click+*Read selected* on the line you want to read. Loaded transfers can then be stored to variables with the command `ci=iiplot;xf=ci.Stack{'Test'};`

2.2.4 Write a script to build a transfer structure

When writing your own script to transcript data to xfstruct format, you must have a MATLAB structure composed at minimum of the fields

- .w : a column vector of frequencies
- .xf : a matrix of measured frequency responses (one row per frequency, one column per measurement channel).

Other fields may be required to specify the type of data and the type of model to use for identification. Two main optional fields are presented here:

- .dof field can be used to specify the meaning of each transfer (input and output DOF).

  This field should be set for title/legend generation (this is a label).

  For **correct display of shapes** in feplot, the .dof may be a direct specification of direction in simple cases where the sensors are really oriented in global axes, but in general is just a label for the sensor orientation map stored in a sens.tdof field. See section 2.7 for details on geometry declaration.
  In the example below one considers a MIMO test with 2 inputs and 4 outputs stored as columns of field .xf with the rows corresponding to frequencies stored in field .w. You script will look like

```
ci=idcom;
[XF1,cf]=demosdt('demo2bay xf');% sample data and feplot pointer
out_dof=[3:6]+.02'; % output dofs for 4 sensors in y direction
in_dof=[6.02 3.01]; % 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(:)';
XF1=struct('w',XF1.w, ...   % frequencies in Hz
           'xf',XF1.xf, ... % responses (size Nw x (40))
           'dof',[out_dof(:) in_dof(:)]);
XF1=xfopt('check',XF1);
ci.Stack{'curve','Test'}=XF1;          % sets data
iicom(ci,'submagpha'); % display
ci.Stack{'Test'}.idopt % field now points to ci.IDopt
ci.IDopt.nsna=size(out_dof,1); % Possibly correct number of outputs
```

```
ci.IDopt.recip='mimo';ci.IDopt % Set reciprocity to mimo

cf.def=ci.Stack{'Test'}; fecom('ch35'); % frequency of first mode
```

You can check these values in the `iicom('InitChannel')` tab.

- `.idopt` field should also be filled for **correct identification** using `id_rc`,. For the main data set called `Test` the `.idopt` field is that of the figure which is more easily accessed from `ci.IDopt`. These correspond to the `IDopt` part of the `Ident` tab (see section 2.4). You can also edit these values in a script. For correct identification, you should set

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

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

2.2.5 Data acquisition

The *SDT* does not intend to support the acquisition of test data since tight integration of acquisition hardware and software is mandatory. A number of signal processing tools are gradually being introduced in `iiplot` (see `ii_mmif FFT` or `fe_curve h1h2`). But the current intent is not to use SDT as an acquisition driver. The following example generates transfers from time domain data

```
frame=fe_curve('Testacq');  % 3 DOF system response
% Time vector in .X field, measurements in .Y columns
frf=fe_curve('h1h2 1',frame); % compute FRF
ci=iicom('Curveid');iicom('curveinit','Test',struct('w',frf.X,'xf',frf.H1))
iicom('SubMagPha');
```

You can find theoretical information on data acquisition for modal analysis in Refs. [2][3][4][5][6].

2.3 Pole initialization (IdAlt and IdMain filling)

The first step of the model identification (see the whole process at section section 2.2.2) is to build an initial list of poles. This list can be provided from various ways:

- Using an external algorithm. The list of poles is then manually imported (section 2.3.1)
- Using the LSCF algorithm (section 2.3.2)
- By iteratively adding poles using a single pole estimator (section 2.3.3)

In the GUI, algorithms linked to the pole initialization are grouped under `AddPoles` :

- `e + .01` : Perform single pole estimation around a given frequency with damping of the order of 1%. (section 2.3.3)
- `BandToPole` : Sequential single pole estimation by band (to be implemented in further release section 2.3.4)
- `Stab` : Open the tab associated to the LSCF algorithm to build a stabilization diagram and extract poles. The button `AutoId` opens this tab and automatically performs a pole extraction with default values of the algorithm. (section 2.3.2)

2.3.1 External pole estimation

The iteratively refined model is fully characterized by its poles (and the measured data). The initialization of the model optimization process can thus easily be performed from any external modal identification algorithm.

If the external software or script used to perform the identification is able to save the result in the universal file format, simply load it like described in section section 2.2.1 .

Else, after storing the measured transfers as a curve named `Test` in a iiplot figure (see section 2.2.1), add poles with the command

```
ci.Stack{'IdMain'}.po =[...
            1.1298e+02   1.0009e-02
            1.6974e+02   1.2615e-02
            2.3190e+02   8.9411e-03];
% ci is the pointer to the iiplot figure containing the Test curve
```

where the array contains as many lines as poles : the first column provides the pole frequencies in Hz and the second one the pole dampings.

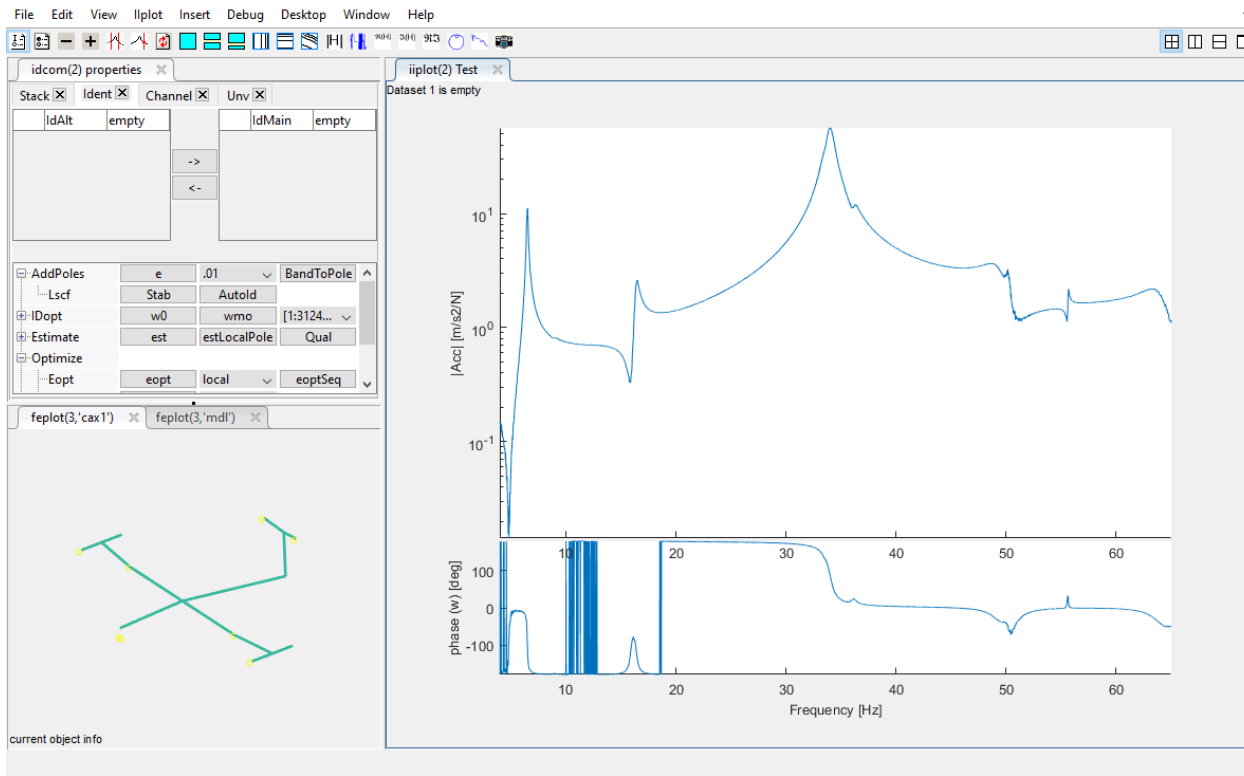
With the list of poles and the measured transfers, you have all you need to recreate an identified model (even if you delete the current one, see section section 2.5) but it also lets you refine the model by adding the line corresponding to a pole that you might have omitted.

2.3.2 LSCF

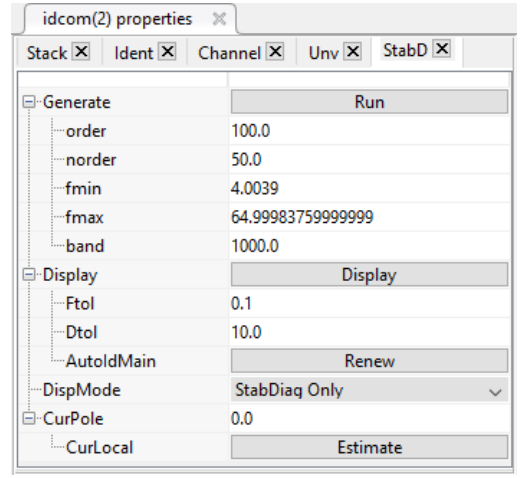
The LSCF algorithm is based a rational fraction description of the transfers. The interest of this algorithm is that polynomials are expressed on the base of the z transform which deeply improves the numerical conditioning (often problematic for high order models in the rational fraction form). Moreover, classical stabilization diagram resulting from the identification at various model orders is often very "clean": numerical modes which either compensate noise or residual terms have negative damping and or thus easily removed from the diagram.

The following tutorial describes how to initialize the poles using the LSCF algorithm.

1. ▶ Execute the command `iicom('dockid')` to open an empty dock and load the wireframe and the transfers contained in the file *SDTPath/sdtdemos/gartid.unv* (Do not load the identification result because it will be performed in the following). See section 2.2.1 for the data loading procedure, or just click on `Run` in the html version of the documentation.



2. ▶ In the tab `Ident`, click on the button `Stab` to open the Tab `StabD` which allows interaction with the stabilization diagram built with the LSCF algorithm.



The button `AutoId` open this StabD tab and directly performs diagram building and pole extraction with default values of the algorithm. It is often useful for a quick evaluation.

3. ▶ The StabD tab contains options to build the stabilization diagram in the sub-list under Generate :

   - order : Maximum order of the model. The order of the model equals the number of poles used to fit the measured data. It is often necessary to select an order significantly higher than the expected number of physical poles in the band because the identification results in many numerical poles which compensate out-of-band modes and noise. Selecting at least ten times the number of expected poles often gives good results according to our experiment.
   - norder : Minimum order to start the stabilization diagram (low model orders often show very few stabilized poles)
   - fmin : Minimum frequency defining the beginning of the band of interest
   - fmax : Maximum frequency defining the end of the band of interest
   - band : Sequential iteration can be performed by band of the specified frequency width. The interest is that in presence of many modes, it is more efficient to perform several identifications by band rather than increasing the model order.

   The building of a stabilization diagram with a maximum order of 100 is not very costly and should be used for most applications. We advise then to estimate the total number of poles in the whole band of interest (fmax-fmin), to divide this total bandwidth by this number and to multiply the result by 5 in order to find the band width which contains in average 5 expected modes (20 times less than the maximum model order).

   In our test case, we attempt to find 12 modes in a total bandwidth of 60Hz) : set the band parameter to 60/12*5=25 Hz.

4. ▶ Click on Generate to build the stabilization diagramm.

   In the diagram, the status of the poles are marked by

   - A red circle when a new poles with positive damping is found
   - A yellow triangle when a consecutive poles are stable in frequency or damping
   - A blue cross when consecutive poles are stable in frequency and damping for since at least 5 consecutive orders

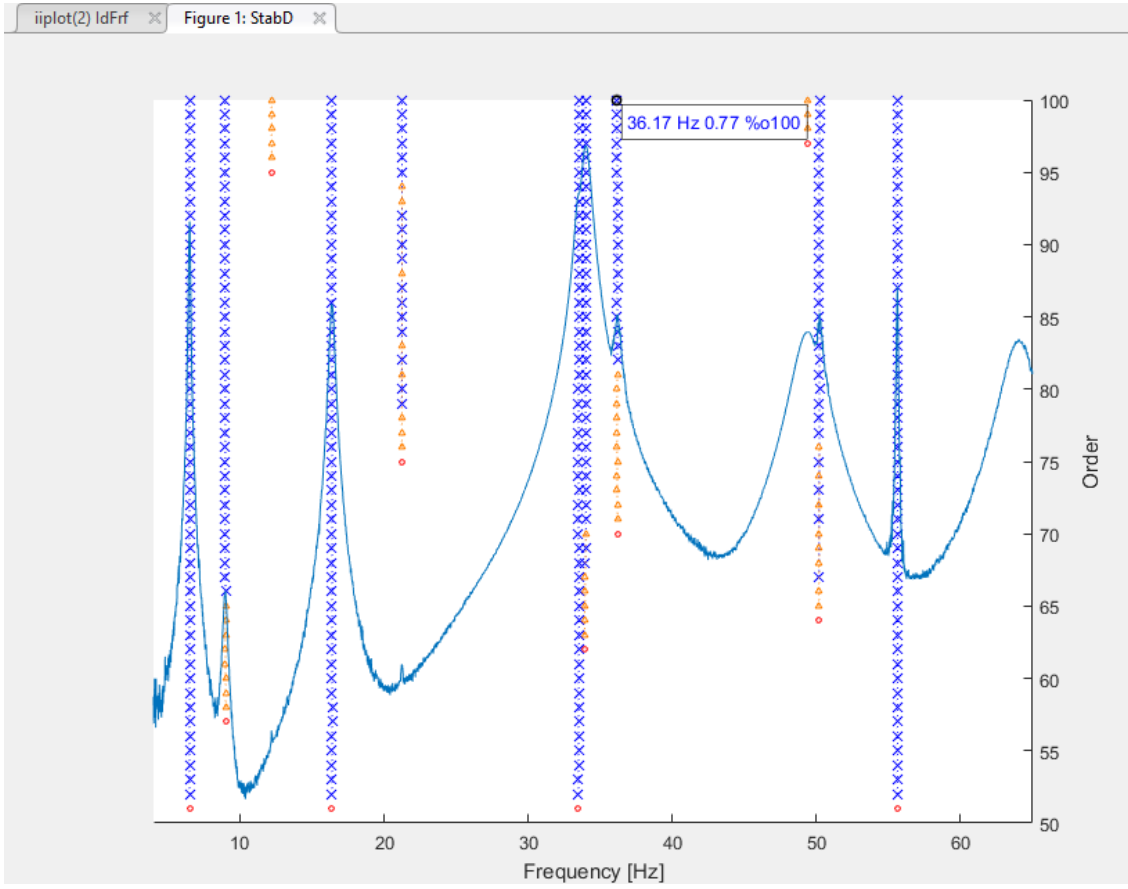
   Frequency and damping stability are defined by the parameters Ftol and Dtol under the sub-list Display. If relative frequency or damping of poles from consecutive model orders are below the parameter values (in %), they are considered stable.

   In presence of very clean measurements of a very strictly linear system, these values could be more restrictive. In the opposite, they should be increase for noisier data and/or in presence

of small non-linearities. When the values of `Ftol` and `Dtol` are modified, click on `Display` to refresh the diagram.

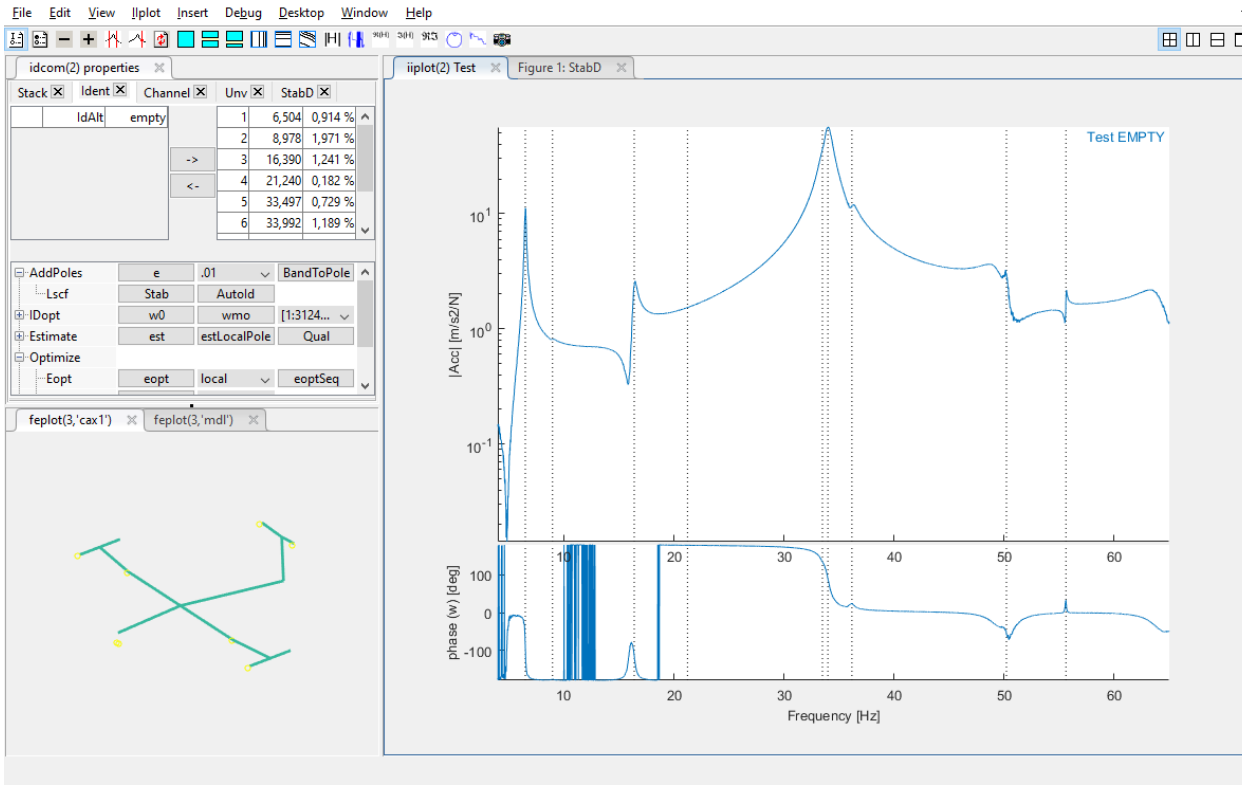
To improve the analysis of the stabilization diagram, mode estimators can be displayed on top of it : the list of all available mode estimators at the right of `DispMode` (see `ii_mmif` for details)

The stabilization diagram displayed with the `logSumI` mode estimator leads to this picture.



5. To automatically extract all stabilized poles (with a blue cross at the last model order), click on `Renew` at the line `AutoIdMain`. The button specifies "Renew" because all current poles in the fmin - fmax band will be deleted and replaced by the extracted ones from the diagram.

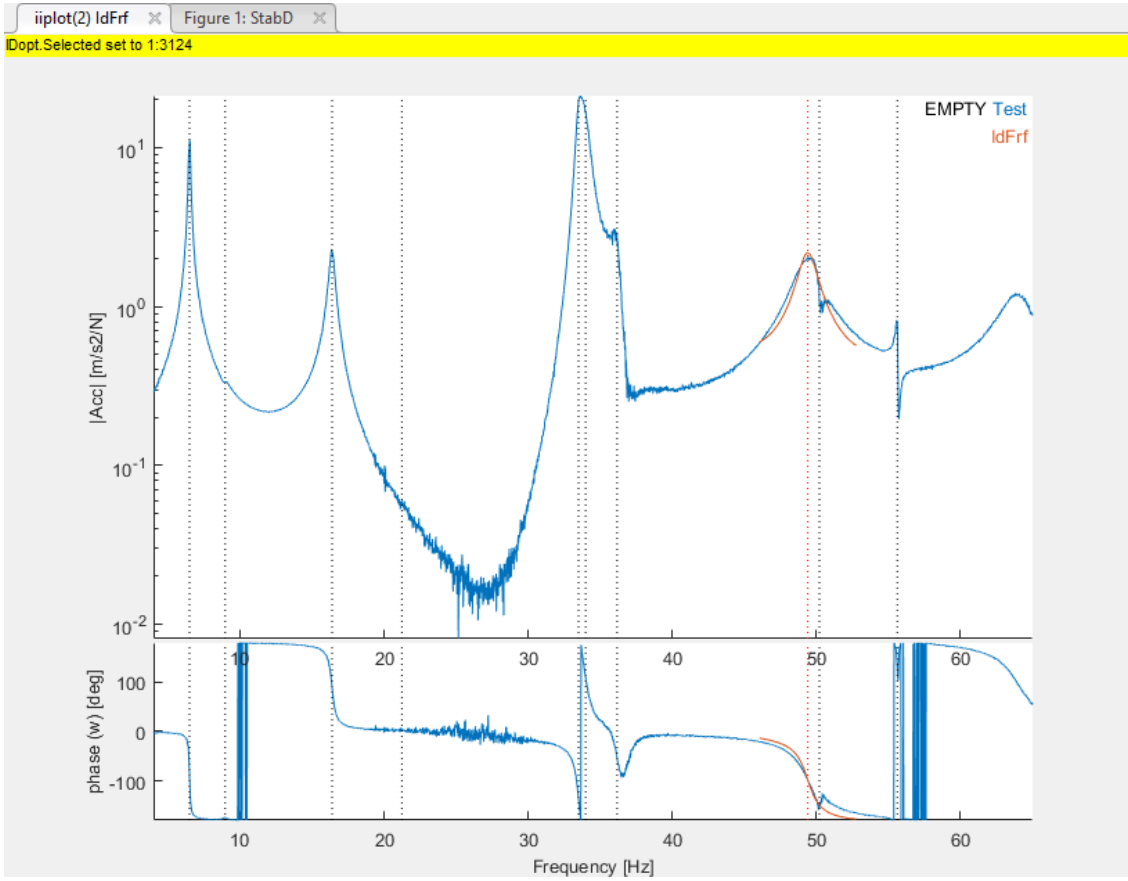
   The extracted poles are displayed at the right table of the tab `Ident`. On the transfers, pole locations are specified by the vertical lines.



6. ▶ Back to the stabilization diagram, two columns are started but not stabilized around 12Hz and 50Hz. For the column at 12Hz, the logSumI indicator shows almost no resonance. For the column at 50Hz, the resonance is well visible but more damped than the close mode.

   To evaluate the pertinence of the poles despite that they do not fully satisfy the stabilization criteria, click on the icon and select the last order of the column.

   In the `StabD` tab, click on `Estimate` at the line `CurLocal`. This action performs a local estimation with the selected pole around its frequency. The channel presenting the highest contribution for this mode is automatically selected and the synthesized transfer is superposed to the measurement.



The synthesized transfer does not exactly fit the measurements (which is very noisy around this frequency) but is enough representative to be selected as initial pole prior to optimization.

7. The pole used to perform the local estimation is stored in the left table of the tab `Ident` : the list of the alternate poles. Because it is representative enough to describe the mode, it can be added to the list of main poles (the right table) by clicking on the arrow.

   Do the same for the not stabilized column around 12Hz. The result is much more doubtful because the mode is almost not visible and the measurement very noisy. More over the local estimation does not fit very well. Nevertheless, add this pole to the main list: we will analyze its pertinence in the following using Quality criteria and trying to optimize it.

Finally, the mode estimator on top of the stabilization diagram shows that a mode at the right of the frequency band is probably there but not identified by the LSCF algorithm. This case can be handled by manually adding a pole using the single pole estimator.

2.3.3 Single pole estimate

Because getting an initial estimate of the poles of the model is the often tedious, algorithms like LSCF or other broadband algorithms are very helpful to quickly extract most of the poles: dynamic responses of structures typically show lightly damped resonances which are most of the time well detected. Nevertheless, using such algorithm often leads to two issues that need to be handled:

- The poles from some modes visible in the transfer have not been extracted
- Some extracted poles do not correspond to physical modes

To deal with missing poles, the easiest way to enrich the initial estimate of the poles is to use a narrow band single pole estimation near considered resonances 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 `ci.Stack{'IdAlt'}`. Once the estimate found, the `iiplot` drawing axes are updated to overlay `ci.Stack{'Test'}` (the measured transfers) and `ci.Stack{'IdFrf'}` (the narrow band transfer synthesis).

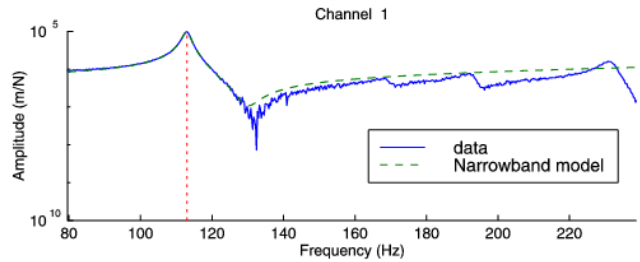


Figure 2.8: Pole estimation.

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 your current model, you can add the estimated pole (add poles in `ci.Stack{'IdAlt'}` to those in `ci.Stack{'IdMain'}`) using the `idcom ea` command (or the associated button : arrow pointing to the right). If the fit is not appropriate you can change the number of selected points/bandwidth and/or the central frequency.

Remark : In the interface or using `idcom e` command, an initial guess of the damping value is used to search for the local mode. The algorithm sometimes fails if this value is too far from the real damping.

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 `ci.Stack{'IdMain'}`.

To deal with cases where you have added too many poles to your current model, use the `idcom er` (or the associated button : arrow pointing to the left) command to 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 2.1).

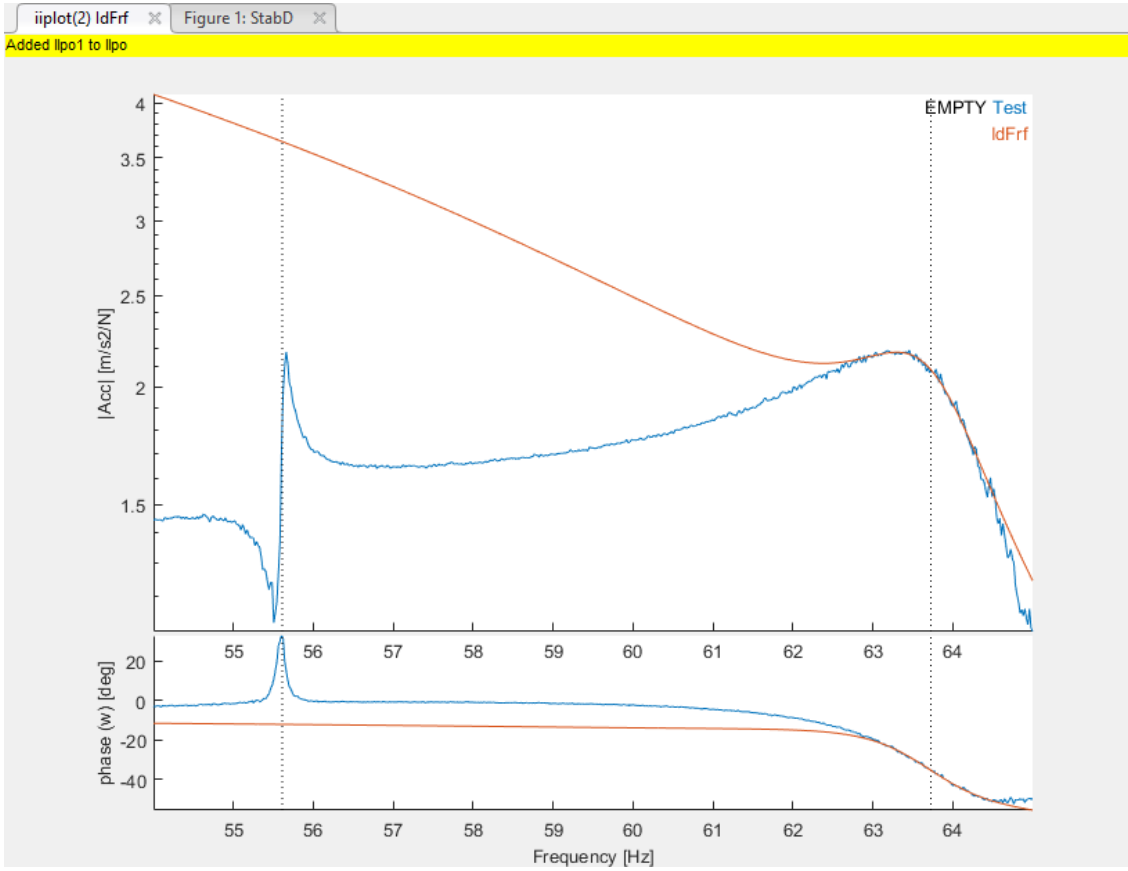
```
gartid % Open interface with gartid demo
idcom('e .1 6')

%idcom('Est 0.1 6.0000); % does click
%LinLS:  2.110e+02, LogLS        Inf, nw 63
% mean(relE) 0.03, scatter 0.16 : good
%Found pole at 6.4901e+00   8.7036e-03
```

Let's go back to the previous tutorial to add the missing pole at the end of the frequency band.

If you have not performed previous tutorial (or if you closed everything at the end), click on ▶ in the HTML version of the documentation to get ready for the following.

8. ▶ Click on the button `e` in the tab `Ident`. Then click approximatively at the location of the resonance to start the single estimation algorithm at that frequency. Please note that, especially in presence of very lightly damped structure, it is sometimes necessary to edit the

value of the expected damping in the list on the right of the button e for the algorithm to find the correct pole.



The fit is correct at the resonance: add the pole to the main list by clicking on the arrow − >

2.3.4 Band to pole estimate

A procedure allowing to add several poles by dragging the mouse to select a band for the single pole estimator will be implemented in further release. Currently the procedure only takes the maximum of the band and does not estimate damping.

2.3.5 Direct system parameter identification algorithm

(Obsolete) A class of identification algorithms makes a direct use of the second order parameteri-

zation. 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. [7]. Constraining the model to have the second-order form

$$\begin{aligned} [-\omega^2 I + i\omega C_T + K_T]\,\{p(\omega)\} &= [b_T]\,\{u(\omega)\} \\ \{y(\omega)\} &= [c_T]\,\{p(\omega)\} \end{aligned} \tag{2.1}$$

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

$$\begin{aligned} \left\{ \begin{array}{c} q' \\ q'' \end{array} \right\} &= \left[\begin{array}{cc} 0 & I \\ -K_T & -C_T \end{array} \right] \left\{ \begin{array}{c} q \\ q' \end{array} \right\} + \left[\begin{array}{c} 0 \\ b_T \end{array} \right] \{u(t)\} \\ \{y(t)\} &= [c_T \;\; 0] \left\{ \begin{array}{c} q \\ q' \end{array} \right\} \end{aligned} \tag{2.2}$$

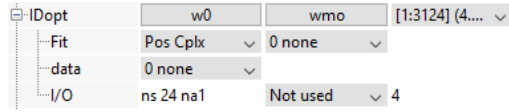
The frequency content of the input $\{u\}$ has a strong influence on the results obtained with id_dspi. Quite often it is efficient to use it as a weighting, rather than using a white input (column of ones) in which case the columns of $\{y\}$ are the transfer functions.

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 2.8.2 .

2.3.6 Orthogonal polynomial identification algorithm

(Obsolete) Among other parameterizations used for identification purposes, polynomial representations of transfer functions (5.31) 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. [8]. 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.

2.4 Identification options



Several options need to be defined in order to well specify the frequency domain on which data must be identified, the type of mesured data, the model used to fit, informations on colocated measurents and how to use them.

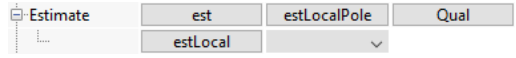
Identification options accessible from the `Ident` tab or from the command line through the pointer `ci.IDopt` (see `idopt` for the full documentation).

Description of the buttons line by line :

- **Idopt** The working frequency band selection specify on which frequencies must the data be identified.
  - *w0* : Resets the working frequency band to the min-max boudaries. This button is similar to clicking on the button and double clicking on the measurements in the iiplot window.
  - *wmo* : Allows to specify min and max frequency by clicking two times at the minimum and then the maximum frequency locations on the measurements in the iiplot window. This button is similar to clicking on the button .
  - *bandwidth history* : Each modification of the working frequency band is stored in this history list and allows to quickly going back to previous selections.
- **Fit** : Several pole/residue models can be used to extract shapes from a list of identified poles, whose complete description can be found in section section 5.6
  - *residue type* : Specify which type of pole/residue model to use : complex mode residues with symmetric pole structure, complex mode residues with asymmetric pole structure or normal mode residues with symmetric pole structure.
  - *residual terms* : To takes into account the influence of out of band modes, residual terms should be used.
- **data** : Specify if the measured transfers are of type displacement/force, velocity/force or acceleration/force
- **I/O** : Information on colocated measurements are needed to enforce the constraint of reciprocity (see section 2.8.2) using the `id_rm` algorithm
  - *nsna* : Display to check if the number of sensors and actuators is correct (if it is not correct, the .dof table defining inputs and outputs of each transfers should be verified, see `curve Response data`)

– *Recip* : Specify how the colocated informations should be used (see section 2.8.2 and `idopt` for more details)

2.5 Estimate shapes from poles



Once a model is created (you have estimated a set of poles in `IdMain`), the residues need to be computed. The classical way to do so in the litterature is to determine residues on the whole frequency band for the synthesized FRFs stored in `ci.Stack{'IdFrf'}` to be as close as possible to the measured data in the least square sense. This strategy and others using narrow bands are detailed in section section 2.5.1 .

To analyze the quality of the identification, several criteria definined by mode and by transfer have been developped to help navigate through the data. The quality table and its analysis are described in section section 2.5.2 .

A non exhaustive list of classical issues using the `id_rc` algorithm is given in section section 2.5.3 .

2.5.1 Broadband, narrowband, ... selecting the strategy

The standard estimation of residues on the whole frequency band is performed with the command `idcom est` (or the equivalent button in the interface).

This method can give good results if the measurements are very clean and the system very close to a perfectly linear system. If noise, non-linear distorsion badly identified pole is present at some frequency bands, especially if it worresponds to high amplitudes in the transfers, fitting all modes together on the whole frequency band can engender strong bias in the identification of residue with low amplitude.

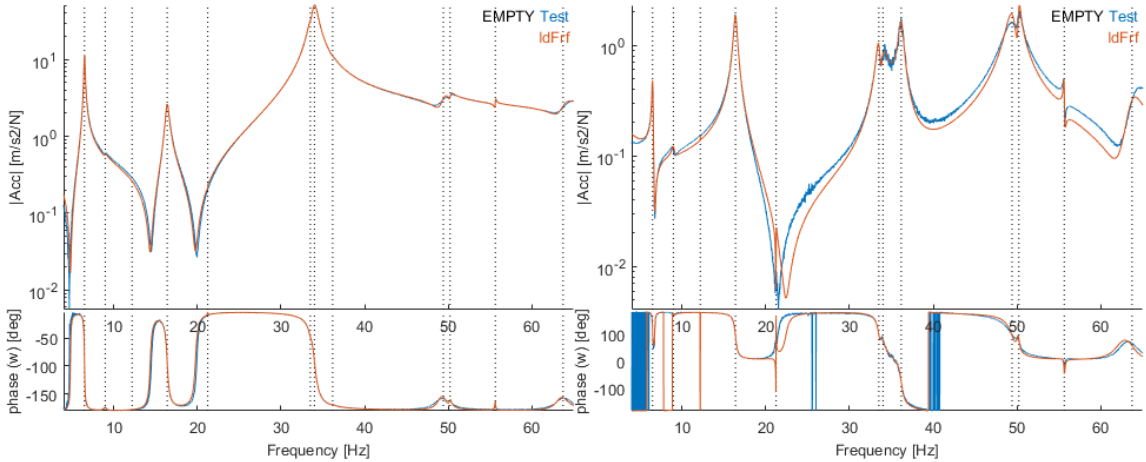
In this case, and if a broadband model is not necessary, it is most of the time preferable to perform a sequential identification with a narrow band arround each mode to extract the residuals. This is automaticaly achived using the command `idcom estl`ocalpole (or the equivalent button in the interface).

An alternative way to handle these problems of bias for some modes is to perform local identifications which update residues only on a smaller working frequency band. To do so, you need to select a close frequency band inside which the residues are poorly identified with the button and then use the command `idcom estl`ocal (or the equivalent button in the interface).

To highlight the differences between these strategies, the following tutorial uses the GARTEUR test case with the initial poles identified in the previous section section 2.3 .

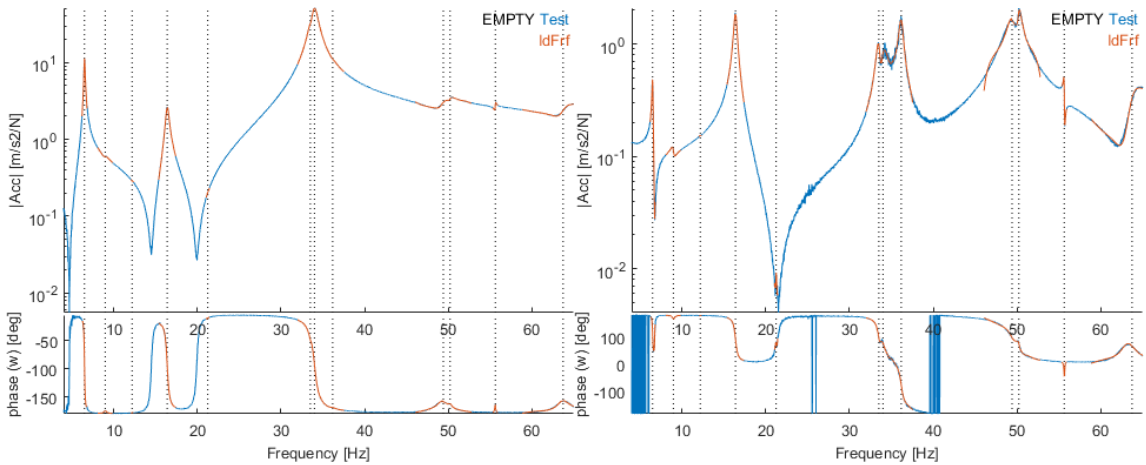
1. ▶ Click on the link in the HMTL version to initialize the tutorial. Else, execute the command `sdtweb('_tuto','gartid')` to open the list of tutorials and execute the first step of the tutorial `Estimate`.

2. ▶ In the `Ident` tab, click on the button `est` to identify the residues using the broadband method.

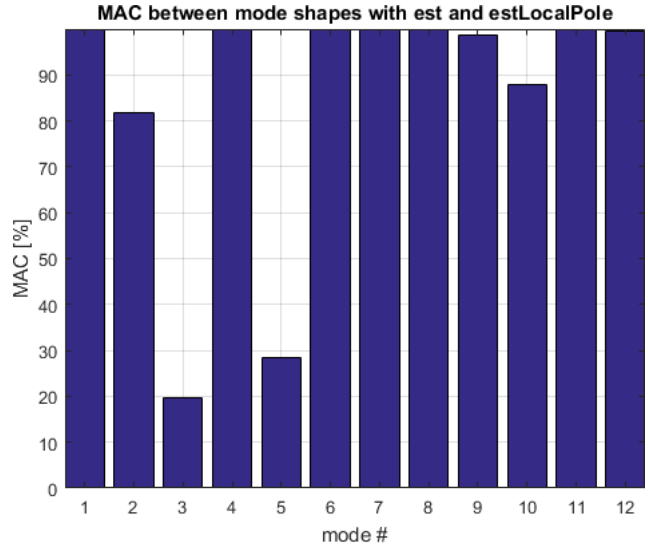


For some transfers the superposition seems quite good like for the first figure whereas it is clearly bad for many modes for some others like the second figure.

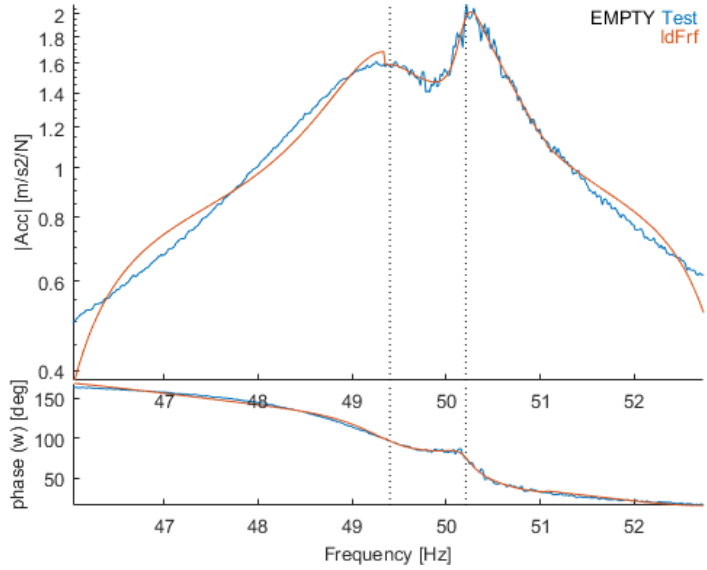
3. ▶ In the `Ident` tab, click on the button `estLocalPole` to identify the residues using the sequential narrowband method.



Each local identification is clearly closer to the measurements than using the broadband strategy. It should be noted that residues correspond to mode shapes and that consequences on proper identification of shapes can be important. The figure below shows the MAC between the set of mode shapes obtained with the `est` versus the `estLocalPole` algorithms.



The two modes 3 and 5 which are very less excited (the physical meaning of these poles is even still question for the moment) are very impacted. Modes 2 which is less excited is quite different. Mode 10 is well visible but the pole seems badly identified as shown on the figure below (zoom on modes 9 and 10) : the residues are differently biased to compensate in the two strategies.



2.5.2 `Qual`: Estimation of pole and shape quality

The need to add/remove poles is determined by careful examination of the match between the test data `ci.Stack{'Test'}` and identified model `ci.Stack{'IdFrf'}`. For a very small amount of data, you could take the time to scan through different sensors, look at amplitude, phase, Nyquist, ... but when the number of sensors and the number of modes become high, the manual scanning is too much time consuming.

Too help navigate through a large amount of data to efficiently analyze the quality of the measurements, several criteria have be defined and can be used to sort sensors by mode. In the following, each pair of sensor/actuator corresponding to a column of the measured transfers H_{Test} associated to a column of the synthesized transfers H_{id} will be indexed by c.

A perfect identification is obtained if measured and synthesized transfers are perfectly superposed. Because the contribution of a mode is characterized by the fact that its amplitude is maximum around the resonance frequency, a classical method to analyze the quality of the fit is to compare the measurement and the identification around each mode. We thus define the **identification error** for a mode j and input/output pair c by

$$e_{j,c} = \frac{\int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c}(s) - H_{id,c}(s)|^2}{\int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{id,c}(s)|^2} \tag{2.3}$$

with ω_j the modal frequency and ζ_j the modal damping. α is a scale factor of the frequency bandwidth, with $\alpha = 1$ corresponding to the classical bandwidth at -3dB and $\alpha = 5$, a pertinent

value used here. This error criterion can be seen as a numerical evaluation of the quality of the historical "circle fit" method. The figure 2.9 shows a simple case on the mode at 4050Hz. On the left, the measurement in blue line is noisy so that the correspondence with the identification in red dotted line is not good. This is coherent with the value of the error criterion evaluated at 30%. On the right, the resonance of the mode is well visible and the superposition with the identification is almost perfect. This visual analysis is well confirmed by the error criterion evaluated at 0.4%

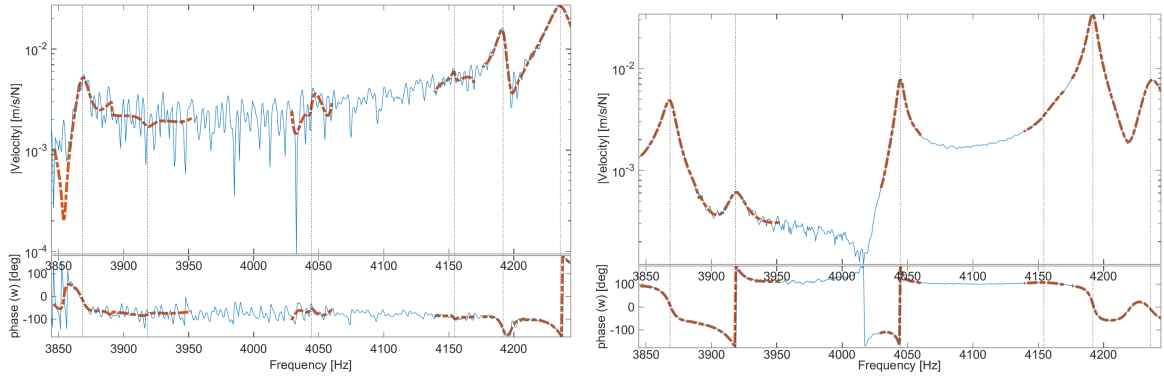


Figure 2.9: Transfer function examples with a high (30%, at left) and low (0.4%, at right) error criterion

For most applications, high error is expected close to vibration nodes where the observability is weak. To avoid taking into account such transfers as badly identified, the **level** criterion for a given mode j and a given sensor/actuator pair c is defined as the ratio between the quadratic mean for the channel c around the resonance and the maximum quadratic mean on all the channels.

$$L_{j,c} = \frac{\int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c}(s)|^2}{\max_c \int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c}(s)|^2} \tag{2.4}$$

Problematic sensors are those presenting a high error despite a significant level. Thus, considering the error criterion and the level criterion is often not appropriate. A new criterion called **Noise Over Signal (NOS)** is obtained by multiplying both criteria together

$$NOS_{j,c} = e_{j,c} \times L_{j,c} \approx \frac{\int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c}(s) - H_{id,c}(s)|^2}{\max_c \int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c}(s)|^2} \tag{2.5}$$

in order to highlight transfers where high error is associated to a un level, and thus critical. For a reasonable identification, the approximation made on (2.5) use the fact that $H_{Test,c}$ et $H_{Id,c}$ should be close and so that $\int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c}(s)|^2 / \int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{id,c}(s)|^2 \approx 1$. This approximation illustrate that the product $e_{j,c} \times L_{j,c}$ is close to the ratio of the identification error (hence a estimation of the noise) over the

maximum response (hence the signal level), which explains the origin of the NOS terminology.

The figure 2.10 (first) shows an example of a transfer function with a high NOS value (8.3%) : the error is very high at 40.4% whereas the level is still significant at 20.5%. The mode is very badly identified (barely visible on this transfer) but the amplitude of the identified residue is important for the definition of the mode shape. The existence of sensor/actuator pairs with high noise level at high amplitude, highlighted by NOS, is typical of weekly excited modes (the controllability is weak for the chosen excitation location). On the second image, the transfer function also shows a high NOS value (24.7%) and a high error (24.7%) but graphically, the mode is very visible. The high NOS value is here due to a bad identification of the pole, which induces a bias in the residue to compensate. This second example illustrates that this criterion is also well adapted to the detection of problems of coherence between measurements (different settings between measurement systems, behavior evolution of the system during measurement,...).

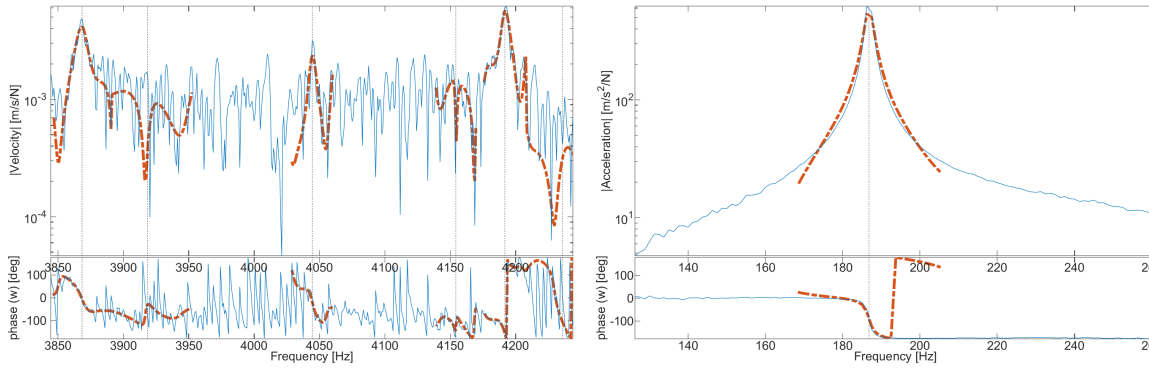


Figure 2.10: Examples of transfer functions showing high NOS values induced by a weak excitation (left) and a bad poles identification (right)

After manual analysis of many measurements, two intermediate cases are often found: the measurement is noisy but still has a sufficient contribution to be identified with confidence or the contribution of a mode is so weak that it cannot be separated from other modes without raising questions on a more or less important estimation bias. To distinguish the two cases, a last **contribution** criterion is introduced

$$C_{j,c} = 1 - \frac{\int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c} - H_{id,j,c}|^2}{\int_{\omega_j(1-\alpha\zeta_j)}^{\omega_j(1+\alpha\zeta_j)} |H_{Test,c}|^2}. \tag{2.6}$$

to measure the modal contribution of a specific mode j relatively to the global response of all the other modes around its resonance frequency, thus giving an indication of its *visibility* ($H_{id,j,c}$ is the transfer synthesis containing only the mode j). For highly noisy transfer functions, this indicator can be negative and is then set to 0.

Figure 2.11 shows transfer functions for which this kind of question is raised. On the first image, around 4050Hz, the mode is well visible despite a relatively high noise level. It could be useful to keep this channel to well interpret the correlation. On the second image, a transfer function is shown where the error is very low but for which the resonance of the considered mode is not visible at all. The capacity to identify the residue with confidence is low because the identification could clearly be significantly biased

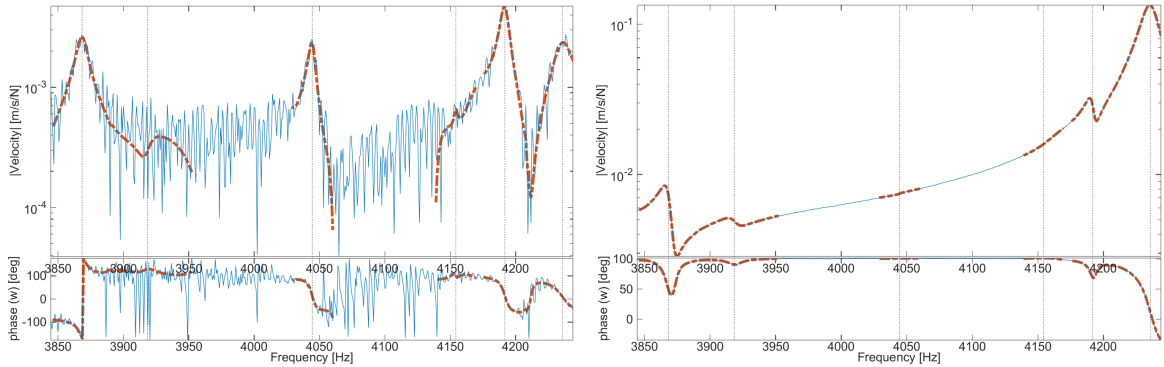


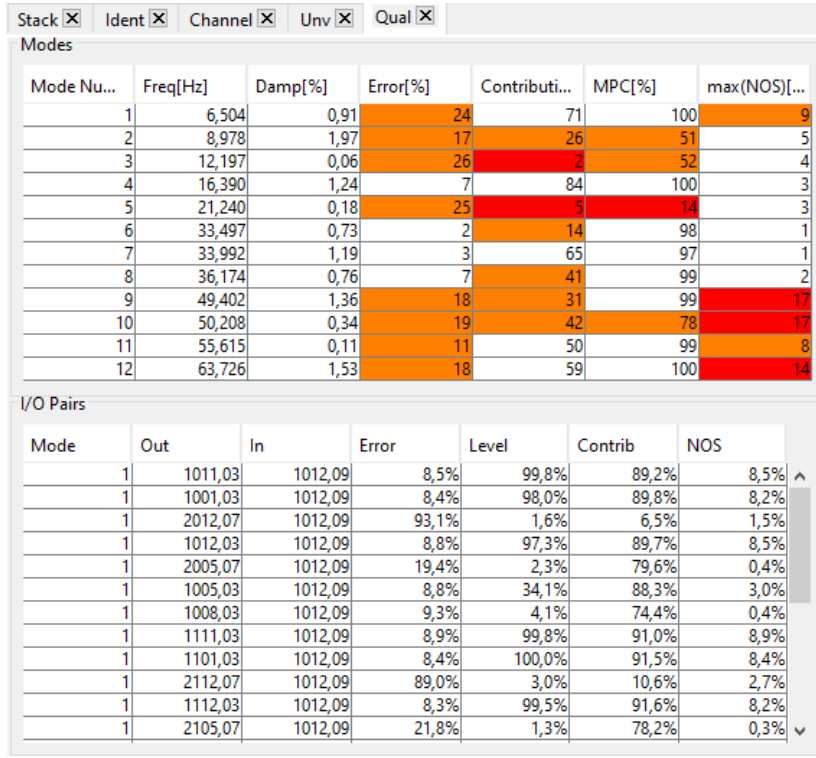
Figure 2.11: Examples of transfer functions: High error of 18.7% with a high contribution of 73.5% (left) and low error of 0.1% with a low contribution of 0% (right).

Proposed criteria allow decomposing identification error sources in contributions by mode and by transfer function (sensor/actuator pair). For each mode, clearly problematic sensors showing high error with low contribution and a low level can be automatically discarded and only results properly identified can be kept with a high confidence on the quality.

Intermediate results can be analyzed in more details using sorting by level, contribution or NOS to highlight problematic transfer functions, as illustrated in the following tutorial.

Let's go back to the previous tutorial. If you have not performed it (or if you closed everything at the end), click on ▶ in the HTML version of the documentation to get ready for the following.

4. ▶ In the `Ident` tab, click on `est` to perform an broad band identification of the residues. Click then on the button `Qual` to open the tab `Qual` which synthesizes all the quality criteria defined above.



Stack | Ident | Channel | Unv | Qual

Modes

Mode Nu...	Freq[Hz]	Damp[%]	Error[%]	Contributi...	MPC[%]	max(NOS)[...
1	6,504	0,91	24	71	100	9
2	8,978	1,97	17	26	51	5
3	12,197	0,06	26	2	52	4
4	16,390	1,24	7	84	100	3
5	21,240	0,18	25	5	14	3
6	33,497	0,73	2	14	98	1
7	33,992	1,19	3	65	97	1
8	36,174	0,76	7	41	99	2
9	49,402	1,36	18	31	99	17
10	50,208	0,34	19	42	78	17
11	55,615	0,11	11	50	99	8
12	63,726	1,53	18	59	100	14

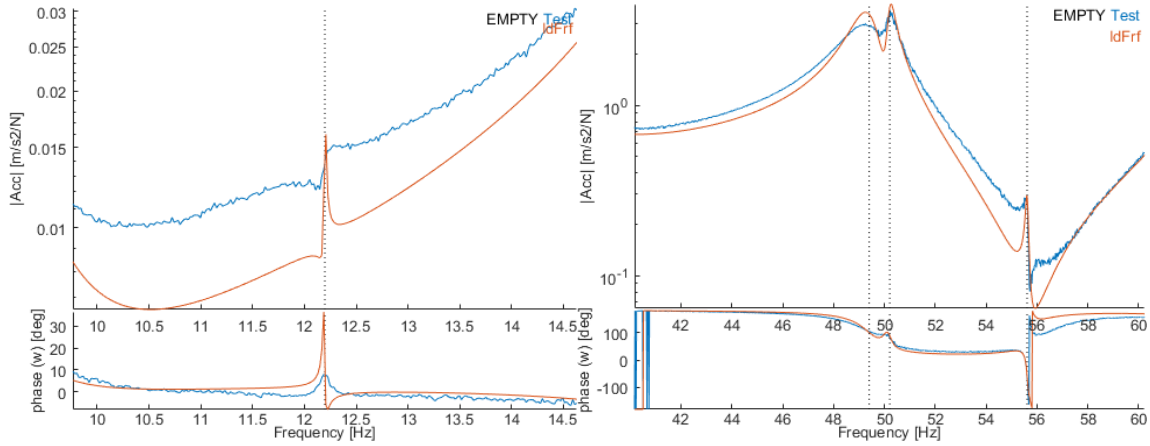
I/O Pairs

Mode	Out	In	Error	Level	Contrib	NOS
1	1011,03	1012,09	8,5%	99,8%	89,2%	8,5%
1	1001,03	1012,09	8,4%	98,0%	89,8%	8,2%
1	2012,07	1012,09	93,1%	1,6%	6,5%	1,5%
1	1012,03	1012,09	8,8%	97,3%	89,7%	8,5%
1	2005,07	1012,09	19,4%	2,3%	79,6%	0,4%
1	1005,03	1012,09	8,8%	34,1%	88,3%	3,0%
1	1008,03	1012,09	9,3%	4,1%	74,4%	0,4%
1	1111,03	1012,09	8,9%	99,8%	91,0%	8,9%
1	1101,03	1012,09	8,4%	100,0%	91,5%	8,4%
1	2112,07	1012,09	89,0%	3,0%	10,6%	2,7%
1	1112,03	1012,09	8,3%	99,5%	91,6%	8,2%
1	2105,07	1012,09	21,8%	1,3%	78,2%	0,3%

The identification quality is globaly poor, with a mean **error** quite high arround most modes. Two modes show a very low mean **contribution** (3 and 5), four modes show a bad **MPC** whereas expected modes are real (2,3,5 and 10) and finally, three modes present a high **max(NOS)** (9 10 and 12).

Clicking on a line of the first table `Modes` updates the second table `I/O Pairs` with the four quality criteria on all sensors for the selected mode. Each criterion can be sorted by clicking on the corresponding column header and clicking on a line perfoms a zoom on the corresponding transfer arround the mode frequency.

This way, we can for example easily zoom on the transfer with the highest **contribution** for the mode 3 and the transfer with the highest **NOS** for the mode 10 :



This highlight the bias in the identification of the residues.

5. Now click in the `Ident` tab on `estlocalpole` to perform a sequential identification by mode with the same poles and click again on `Qual` to update the `Qual` tab.

Stack | Ident | Channel | Unv | Qual

Modes

Mode Nu...	Freq[Hz]	Damp[%]	Error[%]	Contributi...	MPC[%]	max(NOS)[...
1	6,504	0,91	23	69	100	8
2	8,978	1,97	3	23	100	0
3	12,197	0,06	2	0	49	0
4	16,390	1,24	6	85	100	3
5	21,240	0,18	3	2	81	0
6	33,497	0,73	2	14	98	1
7	33,992	1,19	3	67	97	1
8	36,174	0,76	5	40	99	2
9	49,402	1,36	10	35	95	4
10	50,208	0,34	11	37	95	10
11	55,615	0,11	5	41	100	8
12	63,726	1,53	6	46	100	4

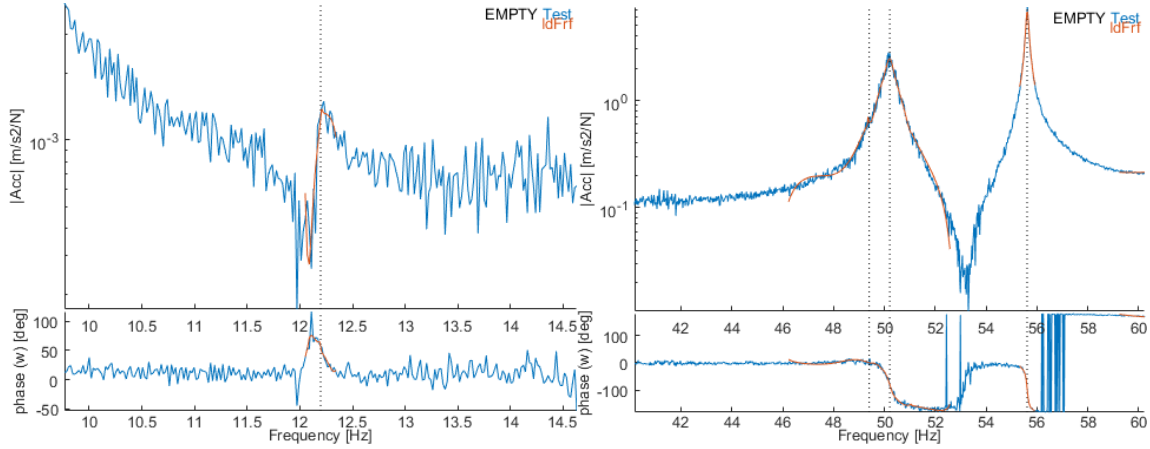
I/O Pairs

Mode	Out	In	Error	Level	Contrib	NOS
1	1008,03	1012,09	8,1%	4,1%	73,8%	0,3%
1	2303,07	1012,09	7,9%	4,7%	91,4%	0,4%
1	2201,08	1012,09	58,1%	0,3%	12,2%	0,2%
1	2012,07	1012,09	92,5%	1,6%	6,8%	1,5%
1	1111,03	1012,09	8,0%	99,8%	91,2%	8,0%
1	2301,07	1012,09	8,1%	5,1%	91,2%	0,4%
1	1012,03	1012,09	7,8%	97,3%	90,0%	7,5%
1	1206,03	1012,09	7,9%	16,8%	91,1%	1,3%
1	3201,03	1012,09	8,3%	23,4%	91,1%	2,0%
1	1205,08	1012,09	73,5%	0,4%	0,0%	0,3%
1	1005,03	1012,09	7,8%	34,1%	87,6%	2,7%
1	1302,08	1012,09	34,0%	0,1%	1,3%	0,0%

The identification quality is clearly better than using the brodband strategy : mean **error** is improved everywhere. Nevertheless, modes 3 and 5 still show very low `contribution` and `MPC`

and mode 10 presents a lower but still high `max(NOS)`.

The zoom on the transfer with the highest **contribution** for the mode 3 and the transfer with the highest **NOS** for the mode 10 can again be displayed :



For mode 3, the resonance is not very visible and the measurement very noisy : this mode is probably not well enough excited and is moreover visible very locally (2 sensors higher than 1% contibution). For mode 10, the high **NOS** do not highlight bad identification anymore (measurement and synthesis are quite well superposed) but shows that the error due to the high measurement noise is present even at sensors where the mode has a high **level** : a better excitation of the mode should reduce the noise and improve the identification quality.

At this step, quality has been evaluated but we are aware that identified poles are possibly biased. Indeed, the strategy of extraction of poles does not use the exact same model than the one used as a second stage to identify the residues. Non-linear optimization of this initial state should be performed and the impact of this optimization on the identification quality is analyzed in Section section 2.6

2.5.3 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.

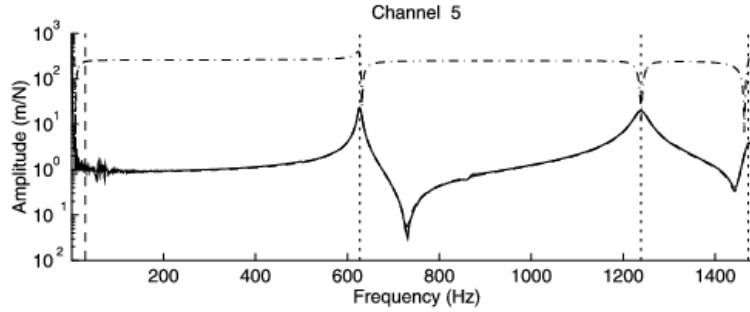


Figure 2.12: Identification problem with low frequency error found for piezoelectric accelerometers

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 (—).

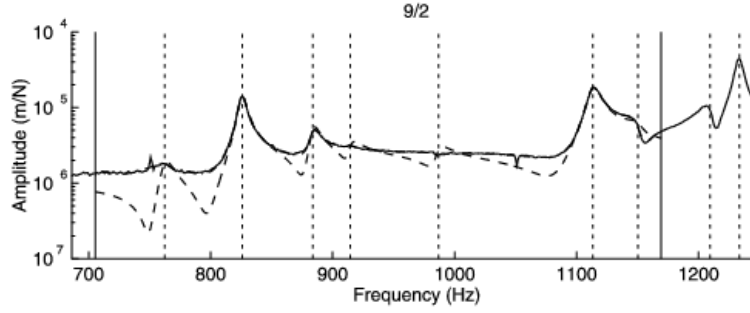


Figure 2.13: Identification problem linked to the proximity of influent out of band modes

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 [9].

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).

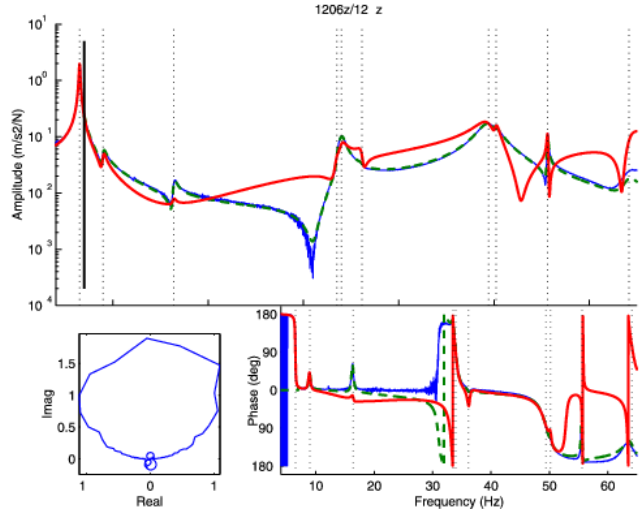
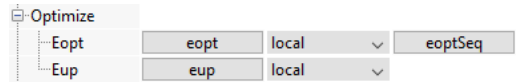


Figure 2.14: Identification problem linked to measurement noise at a major resonance

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).

2.6 Update poles



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 `ci.Stack{'IdMain'}`. This update, which corresponds to a Non-Linear Least-Squares minimization[10][9] which can be performed using different algorithms below.

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).

2.6.1 Eup : for a clean measurement with multiple poles

`idcom eup` (`id_rc` function) starts by reminding you of the currently selected options (accessible from the figure pointer `ci.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 `ci.Stack{'Test'}` and the model with updated poles `ci.Stack{'IdFrf'}`. 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 ' i'` command (for example if you just added a pole that was previously missing).

2.6.2 Eopt : for a band with few poles

`eopt` (`id_rcopt` function) performs a conjugate gradient optimization with a small tolerance to allow faster convergence. But, as a result, it may be useful to run the algorithm more than once. The

algorithm is guaranteed to improve the result but tends to get stuck at non optimal locations.

eup(id_rc function) 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.

You should use the eopt command when optimizing just one or two poles (for example using eoptlocal or 'eopt ' i' to optimize different poles sequentially) or if the eup command does not improve the result as it could be expected.

2.6.3 EupSeq and EoptSeq : sequential narrowband pole updating

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).

A way to limit the divergence issue is to perform sequential local updating arround each pole : one pole is updated at a time so that it is more likely to converge. This sequential optimization as been packaged for both

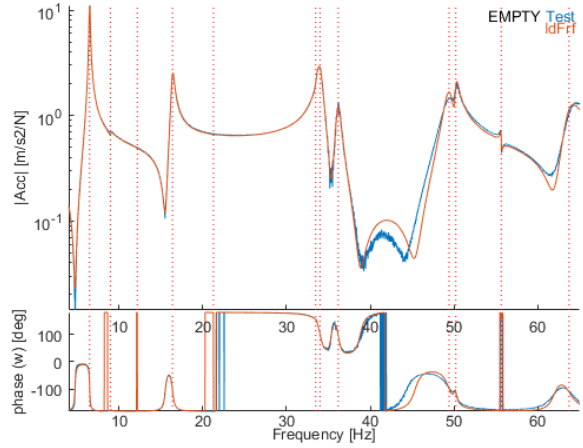
2.6.4 Example for practice

To pratice, the GARTEUR test case already used in previous sections is loaded with an initial set of poles by clicking on ▶.

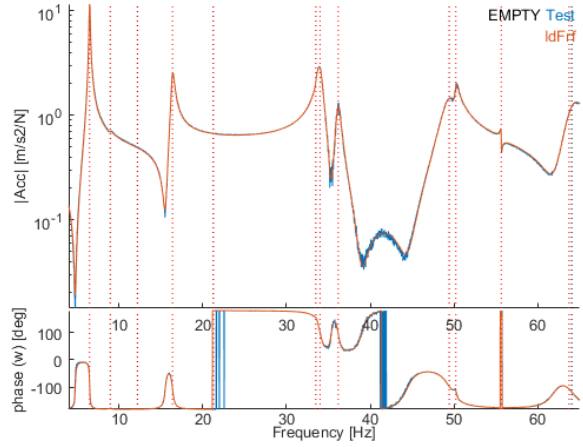
Many strategies can be used to perform the optimization. In the following tutorial, we only propose to guide you through the use of some optimization steps, but the reader is encouraged to test local, broadband, narrowband strategies as he whish to better understand their strengths and weaknesses.

1. ▶ In the Ident tab, click on eopt to perform an broad band optimization (on the selected bandwidth so here on the full bandwidth) using the eopt strategy. Because many poles are present in the band, this algorithm is stuck in a local minimum and the result does not improve much the result.

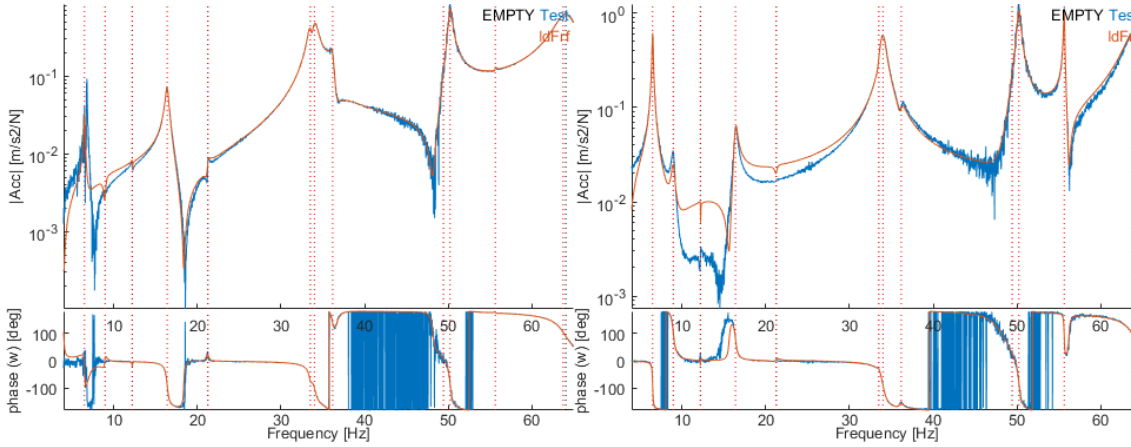
   The figure below shows the transfer and the identification of the sensor 1001.03 (channel 2 in iiplot).



2. ▶ Click now on `eup` to use the other strategy, still on the whole bandwidth. The result deeply improves the identification quality : the same transfer is shown below after the optimization.

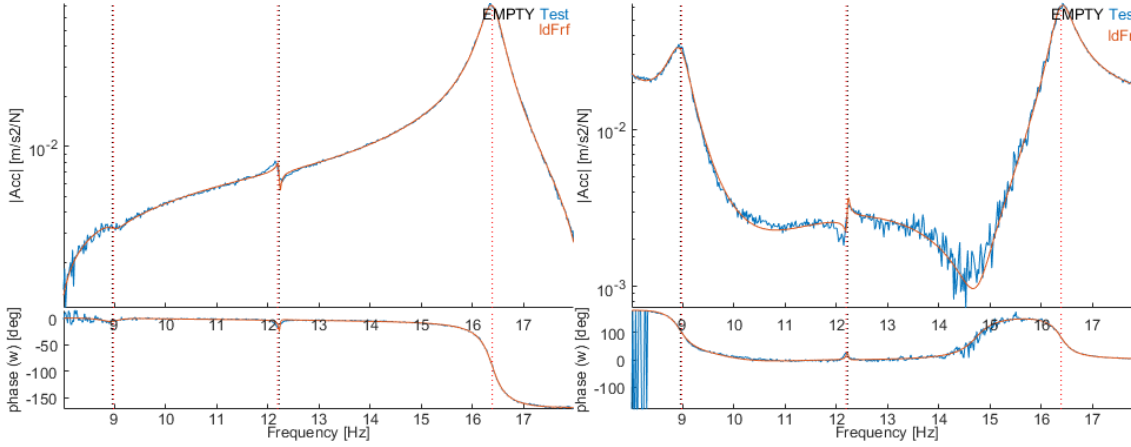


Nevertheless, some transfers still present a quite bad identification, like for instance sensors `2201.08` and `2301.07`.



An interesting observation is that if a smaller band is selected where the fit is poor, without updating the poles, a new identification of the residues may lead to a better identification quality.

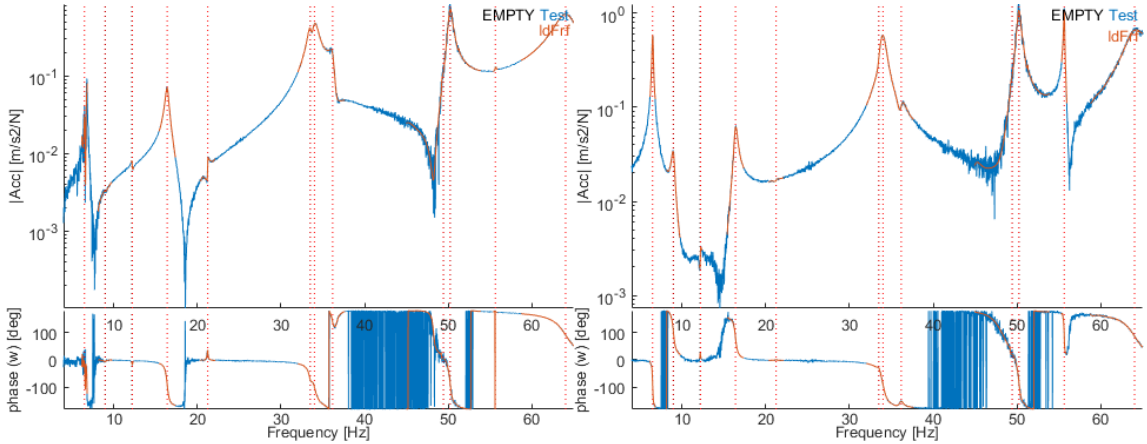
3. Select a narrow band with the button `wmo` between 8 and 18 Hz. Click then on the button `est` to perform a new identification of the residues inside this band without updating the poles. Looking at the same channels as before (sensors `2201.08` and `2301.07`), the fitting quality is clearly improved.



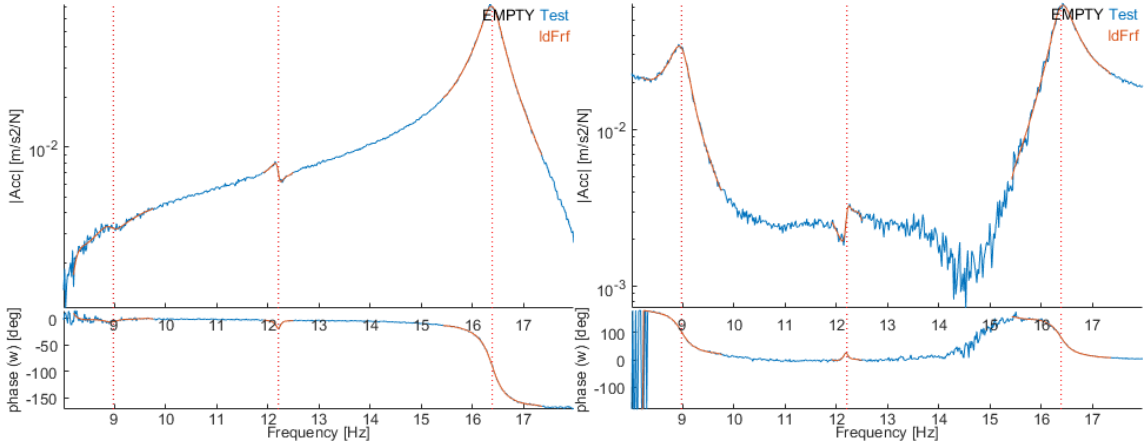
This is due to the fact that taking into account the poles outside this frequency band (especially the noisy first mode) leads to a bias of identification inside this band.

The difficulty is that it is not easy to define which frequency bands can be identified together. To deal with this issue, the sequential local identification of residuals `estlocalpole` can be used. Two version of this strategy have been developped to perform pole updating in addition to residue identification on narrow bands arround each mode : `eoptSeq` and `eupSeq`.

4. Click on `eoptSeq` to perform the sequential optimization. You can perform this optimization several times until convergence if needed.



The vizualisation of the identification on the same band than previously shows a very good fit arround each mode.



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 2.8 .

2.6.5 Background theory

The `id_rc` algorithm (see [10][9]) seeks a non linear least squares approximation of the measured

data

$$p_{\texttt{model}} = \arg\min \sum_{j,k,l=1}^{NS,NA,NW} \left(\alpha_{jk(\texttt{id})}(\omega_l, p) - \alpha_{jk(\texttt{test})}(\omega_l)\right)^2 \tag{2.7}$$

for models in the nominal pole/residue form (also often called partial fraction expansion [11])

$$[\alpha(s)] = \sum_{j\texttt{identified}} \left(\frac{[R_j]}{s-\lambda_j} + \frac{[\bar{R}_j]}{s-\bar{\lambda}_j}\right) + [E] + \frac{[F]}{s^2} = [\Phi(\lambda_j, s)]\,[R_j, E, F] \tag{2.8}$$

or its variants detailed under `res` page 212.

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 2.2.2 . Improved and more robust optimization strategies are still considered and will eventually find their way into the *SDT*.

2.7 Display shapes : geometry declaration, pre-test

Before actually taking measurements, it is good practice to prepare a **wire frame-display** (section 2.7.1 and section 4.1.1 for other examples) and define the sensor configuration (section 2.7.2).

The information is typically saved in a specific `.m` file which should look like the `d_mesh('TutoPre-s3')` demo without the various plot commands. The `d_pre` demo also talks about test preparation.

2.7.1 Modal test geometry declaration

A wire-frame model is composed of node and connectivity declarations.

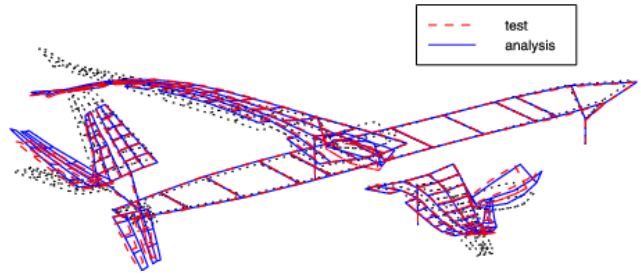


Figure 2.15: Test analysis : wire-frame model.

Starting from scratch (if you have not imported your geometry from universal files). You can declare nodes and wire frame lines using the `fecom Add` editors. Test wire frames are simply groups of beam1 elements with an `EGID` set to -1. For example in the two bay truss (see section 4.1.1)

```
cf=feplot;cf.model='reset';
% fecom('AddNode') would open a dialog box
fecom('AddNode',[0 1 0; 0 0 0]); % add nodes giving coordinates
fecom('AddNode',[3  1 1 0;4  1 0 0]); % NodeId and xyz
fecom('AddNode',[5      0 0 0    2 0 0;
                 6      0 0 0    2 1 0]);
% fecom('AddLine') would add cursor to pick line (see below)
fecom('AddLine',[1 3 2 4 3]);  % continuous line in first group
fecom('AddLine',[3 6 0 6 5 0 4 5 0 4 6]); % 0 for discontinuities
fecom('Curtab:Model','Edit')
%fecom('save')  % will let you save the model to a mat file
feutilb('write',cf.mdl) % generates a script
```

Note that

- `fecom(cf,'AddLine')`, use after node declaration, starts a cursor letting you build the wire-frame line graphically. Click on nodes continue the line, while the context menu allows breaks, last point removal, exit, and display of the commands in the MATLAB command window. This procedure is particularly useful if you already have a FEM model of your test article.
- `fecom(cf,'AddElt')` accessible in the `Model:Edit` tab can be used to add surface or volume elements graphically.
- the `curor:3DLinePick` command in the `feplot` axis context menu is a general SDT mechanism to pick node numbers.
- other GUI based mesh editing tools are described in section 4.4.5 .

- `femesh ObjectBeamLine` and related commands are also typically used to define the experimental mesh (see also `feutil`).
- If you have a FE mesh, you should define the wireframe as a set of sensors, see section 3.1.1 .

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.

2.7.2 Sensor/shaker configurations

The geometry declaration defines fields `.Node` and `.Elt`. The next step is to declare sensors. Once a sensor configuration defined and consistent with input/output pair declarations in measurements (see section 2.2.3), you can directly animate measured shapes (called Operational Deflection Shapes) as detailed in section 2.7.3 . Except for roving hammer tests, the number of input locations is usually small and only used for MIMO identification (see section 2.8).

In the basic configuration with translation sensors, sensor declaration is simply done with a `.tdof` field. Acceptable forms are

- a DOF definition vector (see `mdof`) allows 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.
- a 5 column format (`[SensID NodeID tx ty tz]` giving a sensor identifier (integer or real), a node identifier (positive integer), and the measurement direction in the test mesh axes. This format supports arbitrary orientation.
- a 2 column form `DOF` where each DOF is associated with a local basis, that must be defined in `TEST.bas`.
- the tabular (cell array) definition of sensors and their position, which is more appropriate for large configurations, and is described in section 4.6.2 .

The definition of sensors trough a `.tdof` field is the simplest configuration. For more general setups, see section 4.6 for sensor definitions and section 4.6.4 for topology correlation.

For **interpolation of unmeasured DOFs** see section 3.3.2 .

The following illustrates the first two forms

```
TEST=demosdt('DemoGartteWire');

% simply give DOFs (as a column vector)
TEST.tdof = [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]';

% Transfor to 5 column format, which allow arbitrary orientation
TEST.tdof=fe_sens('tdof',TEST);TEST.tdof

feplot(TEST) % With a .tdof field, a SensDof,Test is defined automatically
fecom('curtab Cases','Test');fecom('ProViewOn')


% You can now display FRFs or modes using
 ci=iicom('curveload gartid'); % load data
 fecom(';ProviewOff;Showline')
% Display FRF
cf.def=ci.Stack{'Test'}; % automatically uses sensor definition 'Test'
% Identify and display mode
idcom('e .05 6.5')
cf.def=ci.Stack{'IdAlt'}; % automatically uses sensor definition 'Test'
```

This new example, mixes all 3 forms

```
cf=demosdt('demogartteplot') % Load data

% simply give DOFs
cf.mdl=fe_case(cf.mdl,'sensdof','Test', ...
[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]');

% Give DOF defined in a local basis
cf.mdl=fe_case(cf.mdl,'sensdof append','Test', ...
[2201.01 1; 3201.03 0; 1206.03 0; 1205.01 1; 1302.01 1]);

% Give identifier, node and measurement direction
cf.mdl=fe_case(cf.mdl,'sensdof append','Test', ...
[1 2301 -1 0 0; 2 1301 0 0 1; 3 2303 -1 0 0; 4 1303 0 0 1]);
fecom('curtab Cases','Test');fecom('ProViewOn')
```

It is also fairly common to glue sensors normal to a surface. The sensor array table (see section 4.6.2) **is the easiest approach** for this objective since it allows mixing global, normal, triax, laser, ... sensors. The following example shows how this can also be done by hand how to obtain normals to a volume and use them to define sensors.

```
% This is an advanced code sample
model=demosdt('demo ubeam');

MAP=feutil('getnormal node MAP',model.Node, ...
  feutil('selelt selface',model)); % select outer boundary for normal

i1=ismember(MAP.ID,[360 365 327 137]); % nodes where sensors are placed
MAP.ID=MAP.ID(i1);MAP.normal=MAP.normal(i1,:);
model=fe_case(model,'sensdof','test', ...
  [(1:length(MAP.ID))' MAP.ID MAP.normal]);

% display the mesh and sensors
cf=clean_get_uf('feplotcf',model);
cf.sel(1)='groupall';cf.sel(2)='-test';
cf.o(1)={'sel2ty7','edgecolor','r','linewidth',2}
```

2.7.3 Animating test data, 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 the response is known at global DOFs no specific information is needed to relate node motion and measurements. Thus any deformation with DOFs will be acceptable. The two basic displays are a wire-frame defined as a FEM model or a wire-frame defined as a `SensDof` entry.

```
% A wire frame and Identification results
[TEST,IdMain]=demosdt('DemoGartteWire')
cf=feplot(TEST);   % wire frame
cf.def=IdMain;     % to fill .dof field see sdtweb('diiplot#xfread')
% or the low level call : cf.def={IdMain.res.',IdMain.dof,IdMain.po}

% Sensors in a model and identification results
cf=demosdt('demo gartfeplot'); % load FEM
```

```
TEST=demosdt('demo garttewire');  % see sdtweb('pre#presen')
cf.mdl=fe_case(cf.mdl,'sensdof','outputs',TEST)

cf.sel='-outputs';  % Build a selection that displays the wire frame
cf.def=IdMain;      % Display motion on sensors

fecom('curtab Plot');
```

When the response is known at sensors that need to be combined (non global directions, non-orthogonal measurements, ...) a `SensDof` entry must really be defined.

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,ci]=demosdt('DemoGartteOds')
cf.def=ci.Stack{'Test'};
% or the low level call :
% cf.def={ci.Stack{'Test'}.xf,ci.Stack{'Test'}.dof,ci.Stack{'Test'}.w}
fecom('CurTab Plot');
```

You can also display the actual measurements as arrows using

```
cf.sens=ci.Stack{'Test'}.dof; fecom ShowArrow; fecom scc1;
```

For a tutorial on the use of `feplot` see section 4.4 .

2.8 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.

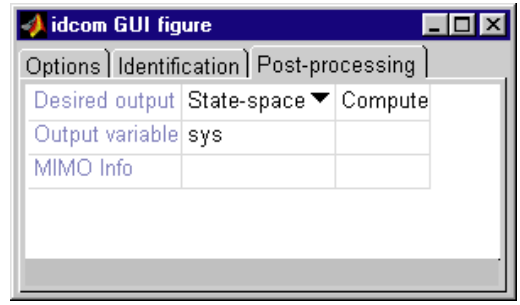


Figure 2.16: idcom interface

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(ci.Stack{'IdMain'})`). For the extra step of state-space model creation use `sys=res2ss(ci.Stack{'IdMain'})`.
`nor=res2nor(ci.Stack{'IdMain'})` or `nor=id_nor(ci.Stack{'IdMain'})` allow transformations to the normal mode form. Finally direct conversions to other formats are given by
`struct=res2xf(ci.Stack{'IdMain'},w)` with `w=ci.Stack'Test'.w`, and
`[num,den]=res2tf(ci.Stack{'IdMain'})`.

These calls are illustrated in `demo_id`.

2.8.1 Multiplicity (minimal state-space model)

Theory

As mentioned under `res` page 212, 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

$$\frac{R_j}{s-\lambda_j} = \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j} \tag{2.9}$$

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 (2.9). 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

$$\frac{R_j}{s-\lambda_j} = \frac{\{c\psi_j\}\left\{\psi_j^T b\right\} + \{c\psi_{j+1}\}\left\{\psi_{j+1}^T b\right\}}{s-\lambda_j} \tag{2.10}$$

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

$$[R_{j1}]_{NS\times NA} = \{U_1\}_{NS\times 1}\,\sigma_1\,\{V_1\}_{NA\times 1}^T \tag{2.11}$$

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 mode shapes 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

$$\begin{gathered}\left(s\,[I]_{2N\times 2N} - \left[\backslash\lambda_j\backslash\right]\right)\{\eta\} = \left[\psi^T b\right]\{u\} \\ \{y\} = [c\psi]\,\{\eta\}\end{gathered} \tag{2.12}$$

which can then be transformed to a real valued state-space model (see res2ss) or a second order normal mode model (see section 2.8.3).

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

```
ci=demosdt('Demo demo_id')
ci.IDopt.nsna=[5 2];  ci.IDopt.reci='no';
RES = id_rm(ci.Stack{'IdMain'},[1 2 1 1]);
% or low level call
[pb,cp,new_res]=id_rm(ci.Stack{'IdMain'}.res,ci.Stack{'IdMain'}.po, ...
     ci.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 2.6 . 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 2.2.2 . 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.

2.8.2 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 (2.9), reciprocity corresponds to the equality of collocated input and output shape matrices

$$\left([c_{\mathtt{col}}]\{\psi_j\}\right)^T = \{\psi_j\}^T [b_{\mathtt{col}}] \tag{2.13}$$

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 $R_{j\mathtt{colS}} = \left(R_{j\mathtt{col}} + R_{j\mathtt{col}}^T\right)/2$. This matrix being symmetric, its singular value decomposition is given by $R_{j\mathtt{colS}} = U_{\mathtt{col}}\Sigma_{\mathtt{col}}V_{\mathtt{col}}^T$ which leads to the reciprocal input and output shape matrices

$$\{c_{\mathtt{col}}\psi_j\} = \left\{\psi_j^T b_{\mathtt{col}}\right\}^T = \sqrt{\sigma_{1\mathtt{col}}}\{U_{1\mathtt{col}}\} \tag{2.14}$$

Typically, there are many more sensors than inputs. The decomposition (2.14) 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 (5.24). The interest of using reciprocal models is to predict non measured transfer functions.

Practice

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

```
ci=demosdt('Demo demo_id')
ci.IDopt.nsna=[5 2]; ci.IDopt.Col=[1 7];
ci.IDopt.reci='mimo';
RES = id_rm(ci.Stack{'IdMain'},[1 1 1 1]);
ci.Stack{'curve','IIxh'}=res2xf(RES,ci.Stack{'Test'}.w); iicom('IIxhOn')
% or low level call
[pb,cp,new_res,new_po]=id_rm(ci.Stack{'IdMain'}.res,ci.Stack{'IdMain'}.po, ...
      ci.IDopt,[1 1 1 1]);
ci.Stack{'curve','IIxh'} = ...
  res2xf(struct('res',new_res,'po',new_po,'idopt',ci.IDopt),ci.Stack{'Test'}.w);
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`/`ci.Stack{'IIxh'}` 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

```
ci=demosdt('Demo demo_id')
[psib,cpsi]=id_rm(ci.Stack{'IdMain'}.res,ci.Stack{'IdMain'}.po, ...
      ci.IDopt,[1 1 1 1]);
uind=3; res_u = (cpsi*diag(cpsi(uind,:))).';
RES=struct('res',res_u,'po',ci.Stack{'IdMain'}.po,'idopt',ci.IDopt);
 ci.Stack{'curve','IdFrf'}=res2xf(RES,ci.Stack{'Test'}.w);
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*).

2.8.3 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 (2.19) 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 (5.4) can be rewritten in a rational fraction form (with truncation and residual terms)

$$[\alpha(s)] = \sum_{j=1}^{NM} \frac{\{c\phi_j\}\left\{b^T\phi_j\right\}^T}{s^2 + 2\zeta_j\omega_j s + \omega_j^2} + [E] + \frac{[F]}{s^2} = \sum_{j=1}^{NM} \frac{[T_j]_{NS\times NA}}{s^2 + 2\zeta_j\omega_j s + \omega_j^2} + E(s) \tag{2.15}$$

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 ci.IDopt.Fit='Normal'. As for the complex residues (5.30), 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

$$[T_j] = \{c\phi_j\}\left\{\phi_j^T b\right\} \tag{2.16}$$

and the dyadic decomposition of the residue matrix can be used as in the complex mode case (see section 2.8.1 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 (2.15) and (2.16) 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 202). This remains true even for multiple modes. A result rarely obtained by other methods.

When a complex mode identification has been performed (ci.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. [12])

$$M = \left(\tilde{\psi}\Lambda\tilde{\psi}^T\right)^{-1}, \quad C = -M\tilde{\psi}\Lambda^2\tilde{\psi}^T M, \quad \text{and} \quad K = \left(\tilde{\psi}\Lambda^{-1}\tilde{\psi}^T\right)^{-1} \tag{2.17}$$

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

$$\sum_{j=1}^{2N} \left\{\tilde{\psi}_j\right\} \left\{\tilde{\psi}_j\right\}^T = \left[\tilde{\psi}\right]_{N\times 2N} \left[\tilde{\psi}\right]^T_{N\times 2N} = [0]_{N\times N} \tag{2.18}$$

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 (2.12). psi2nor then determines c and $\tilde{\psi}$ such that the $\tilde{\psi}$ verify the condition (2.18) 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 [12]. 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

$$\frac{\zeta_j\omega_j\zeta_k\omega_k}{\omega_j\omega_k}^2 \ll 1 \tag{2.19}$$

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

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 obtained with id_nor. A coarse approximation is given by res2nor(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 2.8.3).

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 use of this functionality is demonstrated in the following example.

```
ci=demosdt('demodemo_id') % load data and identify
f=ci.Stack{'Test'}.w;
```

```
nor = id_nor(ci.Stack{'IdMain'});
nor2xf(nor,f,'hz iiplot "IdFrf"'); % Compute response

% compute residual effects and add normal model contributions
res2xf(ci.Stack{'IdMain'},f,ci.IDopt,[5 6],'iiplot "Nor+Stat"');% residues
ci.Stack{'Nor+Stat'}.xf=ci.Stack{'Nor+Stat'}.xf+nor2xf(nor,f,'hz');
iicom('ch1');
```

The normal mode input `nor.pb` and output `nor.cp` matrices correspond to those of an analytical model with mass normalized modes. They can be compared (`ii_mac`) or combined (`fe_exp`) with analytical models and the modal frequencies `nor.freq` and damping matrix `nor.ga` can be used for predictions (see more details in section 3.4).

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. Out of band modes and residual terms are here represented by the $E(s)$ term. Second order models are said to be complete when $E(s)$ can be neglected [13]. The addition of residual terms was illustrated in the example above.

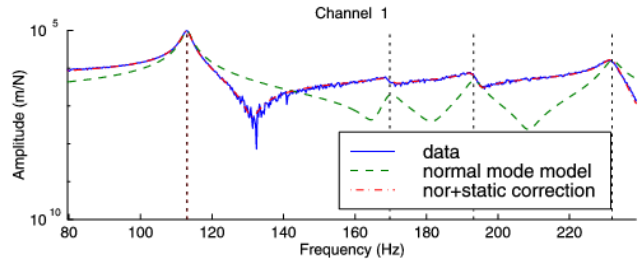


Figure 2.17: FRF xx

3

Test/analysis correlation tutorial

3.1 Topology correlation and test preparation 113
3.1.1 Defining sensors in the FEM model : data handling 114
3.1.2 Test and FEM coordinate systems 117
3.1.3 Sensor/shaker placement 120
3.2 Test/analysis correlation 121
3.2.1 Shape based criteria 121
3.2.2 Energy based criteria 127
3.2.3 Correlation of FRFs 128
3.3 Expansion methods 129
3.3.1 Underlying theory for expansion methods 130
3.3.2 Basic interpolation methods for unmeasured DOFs 131
3.3.3 Subspace based expansion methods 132
3.3.4 Model based expansion methods 134
3.4 Structural dynamic modification 134

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 2.
- 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).

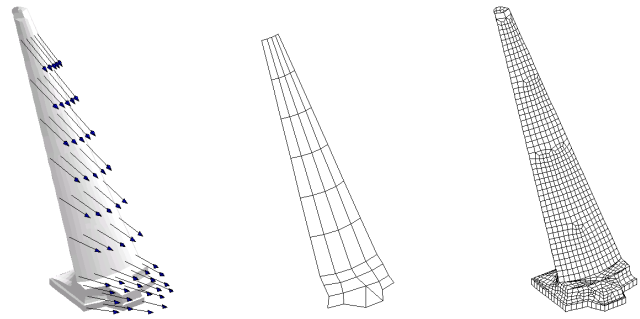


Figure 3.1: FE and wire-frame models

The tools for the declaration of the wire-frame model and of sensor setups are detailed in section 2.7 . Topology correlation and sensor/shaker placement tools are details in section 3.1 . A summary of general tools used to compare sets of shapes is made in section 3.2 . Shape expansion, which deals with the transformations between the wire-frame and FE models, is introduced in section 3.3 . The results of correlation can be used for hybrid models combining experimental and analytical results (see section 3.4) or for finite element model updating (see section 6.5).

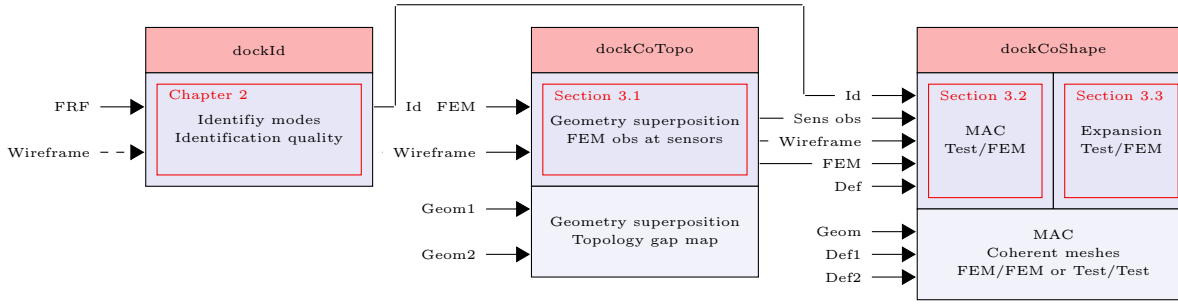


Figure 3.2: Modal identification process with links to corresponding sections

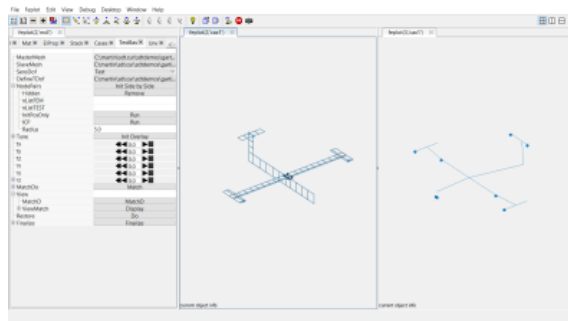
3.1 Topology correlation and test preparation

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`.

Starting with SDT 6.0, FEM sensors (see section 4.6) can be associated with wire frame model, the strategy where the two models where merged is thus obsolete.

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.



3.1.1 Defining sensors in the FEM model : data handling

Two types of data are needed to properly associate a test wire frame model to a FEM :

- a FEM model (see section 4.5). For this simple example, the `FEM` model (stored in `cf.mdl` in the demo) must describe nodes, elements and DOFs
- a test wire-frame model (stored in `TEST` in the demo) with sensors in the `.tdof` field, as detailed in section 2.7.1 for the geometry and section 2.7.2 for sensors

One then declares the wire frame (with sensors) as `SensDof` case entry as done below (see also the `gartte` demo). The objective of this declaration is to allow observation of the FEM response at sensors (see `sensor Sens`).

```
model=demosdt('DemoGartFE') % load FEM
TEST=demosdt('demo garttewire'); % see sdtweb('pre#presen')
% Store Test as SensDof (linked test wireframe) in the FEM
model=fe_case(model,'sensdof','sensors',TEST);
cf=feplot(2); cf.mdl=model; % Display the model in feplot
% Display the superposition of the test wireframe over the FEM
fecom(cf,'ShowFiCoTopo');
% Open the CoShape Dock from cf, already containing needed data
fecom(cf,'dockCoShape');
```

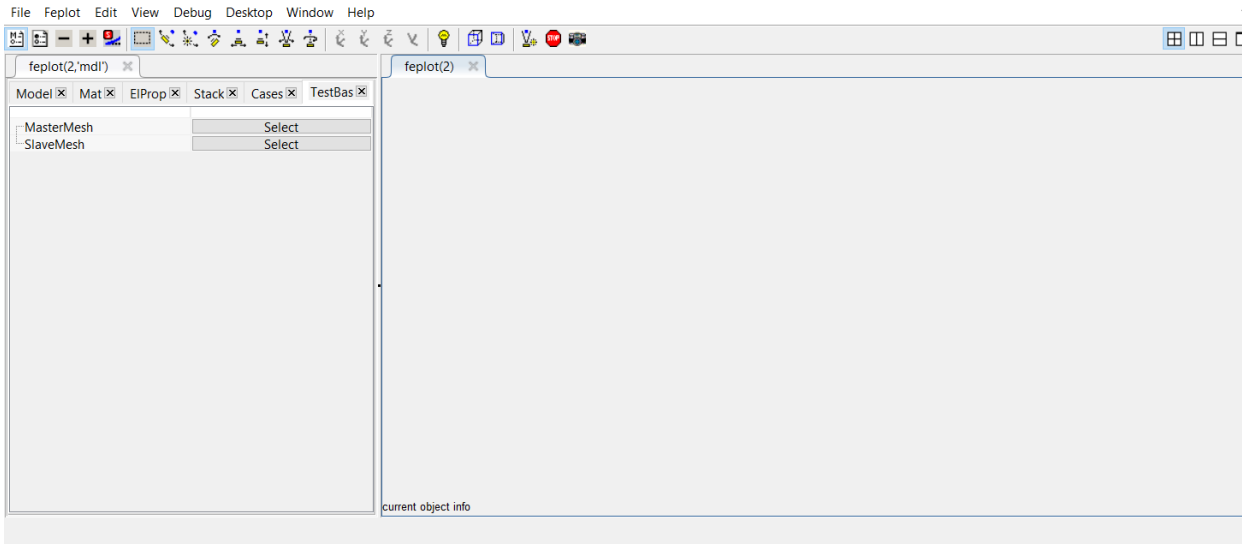
Section 4.6 gives many more details the sensor GUI : the available sensors (`sensor trans`, `sensor triax`, laser, ...). Section 4.6.4 discusses topology correlation variants in more details.

If the data come from files, it can be more convenient to load them directly from the GUI.

Here is a tutorial for interactive data loading in `DockCoTopo` with the `TestBas` tab.

You will need the `garteur` example files, which can be found in *SDTPath/sdtdemos/gart*.m*. If these files are not present, click on the first step on the following tutorial in the HTML version of the documentation or download the patch at the address https://www.sdtools.com/contrib/garteur.zip and unzip the content in the folder *SDTPath/sdtdemos*.

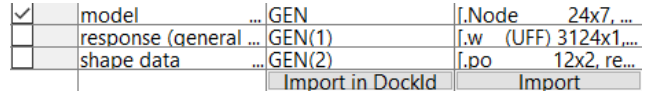
1. Execute the command `fecom('dockCoTopo')` to open an empty dock. You can also click on the button `CoTopo` on the tree in `SDT Root`.



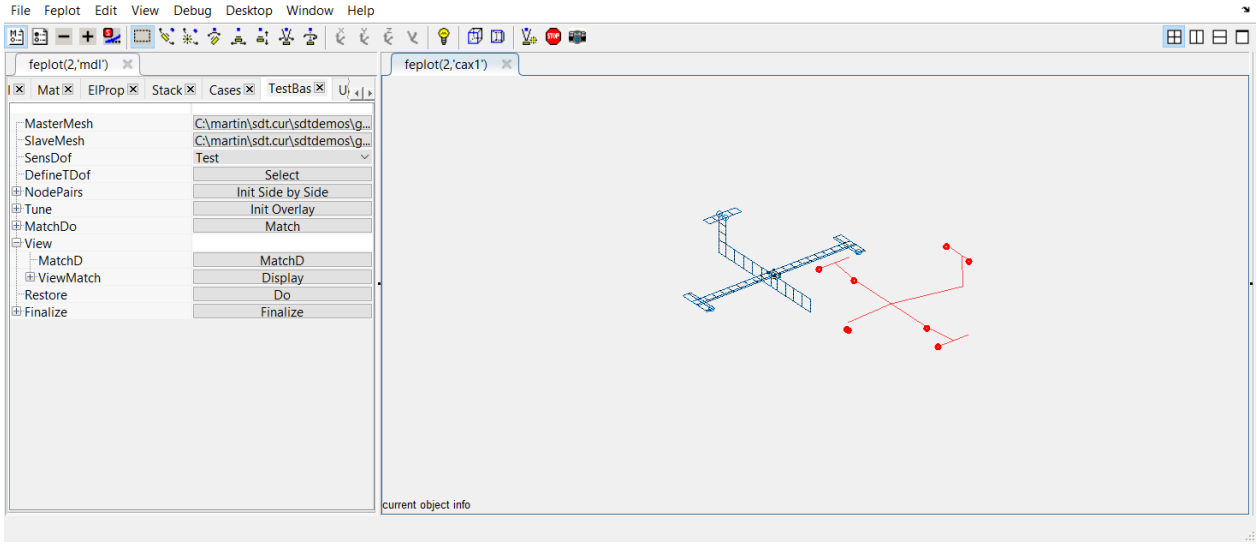
2. Click on `Select` associated to `MasterMesh`. This will open the import model window. Select the file to load : for this tutorial, the file is located at `SDTPath/sdtdemos/gart_mdl.inp`. Data is loaded and displayed in the `feplot` figure.

3. Do the same for the `SlaveMesh`. The test mesh file is located at *SDTPath/sdtdemos/gartid.unv*. Data is loaded and displayed in the `feplot` figure. Once selected, the `Unv` tab is displayed in the `feplot('mdl')` figure : it shows the content of what is inside the `Unv` file.

   Check the box corresponding to `model` and click on `Import`.



   The test wireframe is loaded and displayed in the `feplot` figure in red.

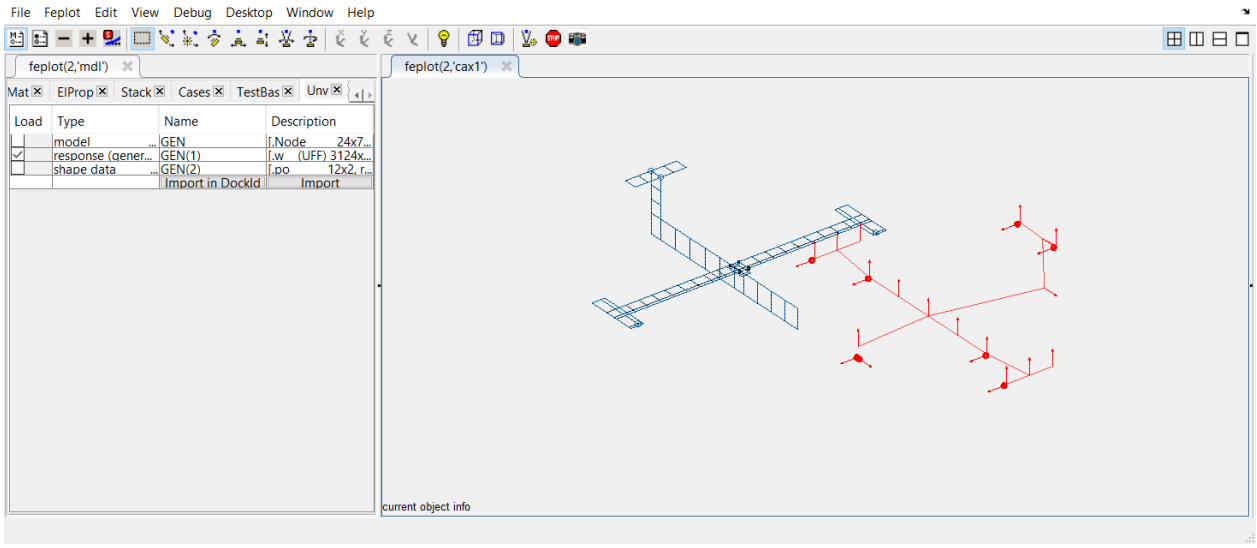


Depending on the loaded data for the the `SlaveMesh`, it contains already or not the sensor definitions : they are shown as red arrows. It is not the case here.

4. To retrieve sensors definition from a `Unv` file, the mesured data need to be loaded.

   Click on `Select` associated to `DefineTDof`. Select again the `Unv` file and in the `Unv` tab, check this time the box corresponding to `response` and click on `Import`.

   The arrows are then built depending on the measured channels (+X,+Y,+Z,-X,-Y,-Z directions associated to each nodes in the geometry), and displayed.



The system coordinate is not the same between the test wireframe and the FEM : the test geometry needs to be moved and superposed to the FEM (this tutorial continues in the following subsection).

3.1.2 Test and FEM coordinate systems

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.

Interactive test mesh placement is available in the SDT GUI, using command `fe_sensGuiTestBas`.

```
% Loading the interactive test mesh placement GUI
 cf=demosdt('demo garttebasis'); % Load the demo data
 cf.CStack{'sensors'} % contains a SensDof entry with sensors and wireframe
 fecom(cf,'setTestBas'); % Open interactive tab in feplot properties
```

Operations permitted through the GUI implementation are available in script commands. The *modus operandi* considers a three steps process.

- Phase 1 is used get the two meshes oriented and coarsely aligned. The guess is more precise if a list of paired nodes on the FEM and TEST meshes can be provided.
- In phase 2, the values displayed by `fe_sens`, in phase 1 are fine tuned to obtain the accurate alignment.
- In phase 3, the local basis definition is eliminated thus giving a `cf.CStack{'sensors'}` entry with both `.Node` and `.tdof` fields in FEM coordinates which makes checks easier.

In peculiar cases, the FEM and TEST mesh axes differ, and a correction in rotation in the Phase 2 may be easier to use. An additional rotation to apply in the TEST mesh basis can be obtained by fulfilling the field `rotation` in Phase 2. The rotations are applied after other modifications so that the user can directly interpret the current `feplot` display. The rotation field corresponds to a `basis rotate` call. The command string corresponding to a rotation of 10 degrees along axis y is then `'ry=10;'`. Several rotations can be combined: `'ry=10; rx=-5;'` will thus first perform a rotation along y of 10 degrees and a rotation along x of -5 degrees. These combinations are left to the user's choice since rotation operations are not symmetric (*e.g.* `'rz=5;rx=10;'` is a different call from `'rx=10;rz=5;'`).

The following example demonstrates the 3 phases in a script.

```
 cf=demosdt('demo garttebasis'); % Load the demo data
 cf.CStack{'sensors'} % contains a SensDof entry with sensors and wireframe
```

```
% Phase 1: initial adjustments done once
 % if the sensors are well distributed over the whole structure
 fe_sens('basis estimate',cf,'sensors');

% Phase 1: initial adjustments done once, when node pairs are given
 % if a list of paired nodes on the TEST and FEM can be provided
 % For help on 3DLinePick see sdtweb('3DLinePick')
 cf.sel='reset';    % Use 3DLinePick to select FEM ref nodes
 cf.sel='-sensors'; % Use 3DLinePick to select TEST ref
 i1=[62 47 33 39;          % Reference FEM NodeId
    2112 2012 2301 2303]';% Reference TEST NodeId
 cf.sel='reset'; % show the FEM part you seek
 fe_sens('basis estimate',cf,'sensors',i1);


%Phase 2 save the commands in an executable form
% The 'BasisEstimate' command displays these lines, you can
% perform slight adjustments to improve the estimate
 fecom(cf,'initTestBas') % When you change a value script below displayed
 fe_sens('basis',cf,'sensors', ...
  'x',      [0 1 0], ... % x_test in FEM coordinates
  'y',      [0 0 1], ... % y_test in FEM coordinates
  'origin',[-1 0 -0.005],... % test origin in FEM coordinates
  'scale', [0.01]); % test/FEM length unit change

%Phase 3 : Force change of TEST.Node and TEST.tdof to FEM coordinates
fecom('SetTestBas',struct('BasisToFEM','do'));
fe_case(cf.mdl,'sensmatch')
sens=fe_case(cf.mdl,'sens')
```

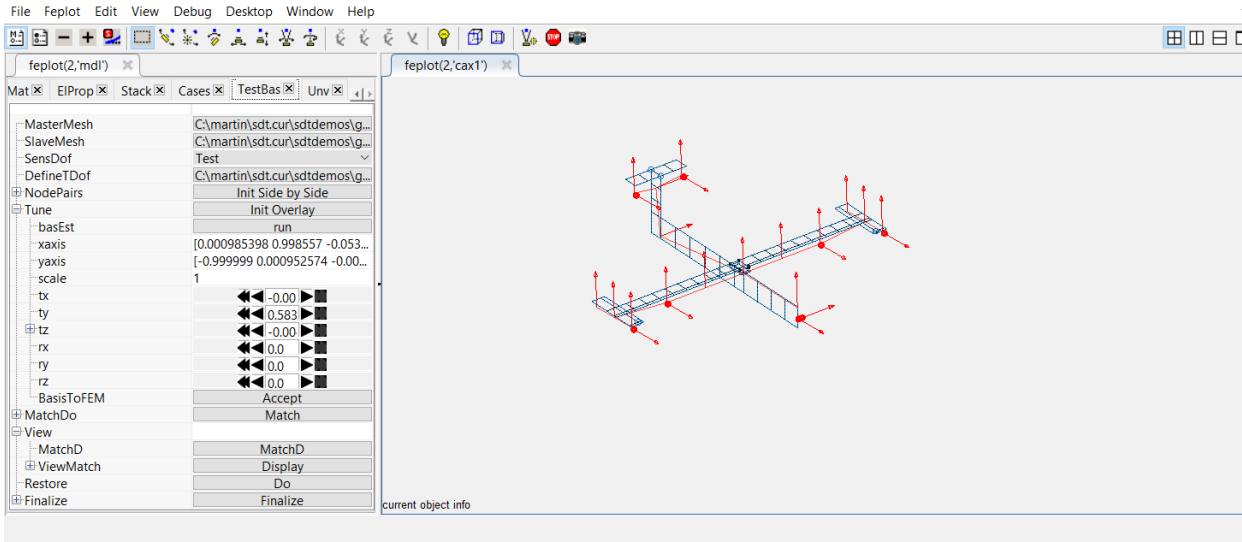
Note that FEM that use local coordinates for displacement are discussed in `sensor trans`.

Here is the continuation of the tutorial for interactive way to superpose and match sensors over the FEM.

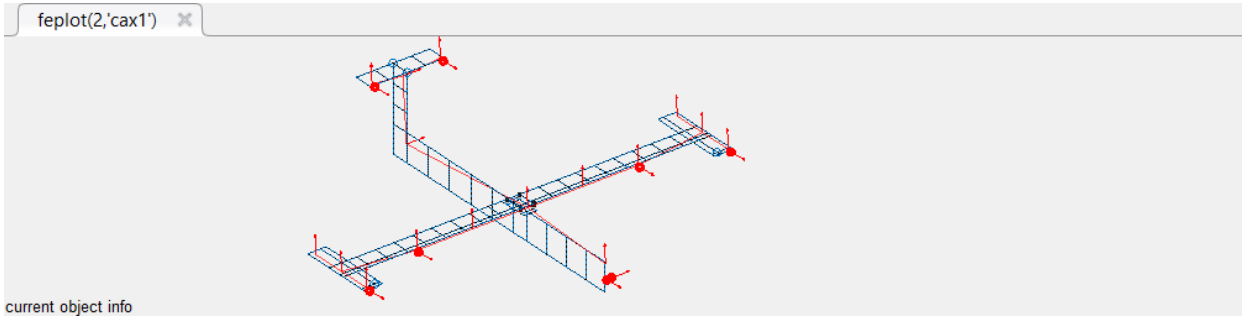
If you have not performed previous tutorial (or if you closed everything at the end), click on this link in the HTML version of the documentation to get ready for the following.

5. To begin with, it is often useful, if the test geometry globally describes well the model geometry, to perform an automatic initial guess for the superposition. To so so, click on the button `run`

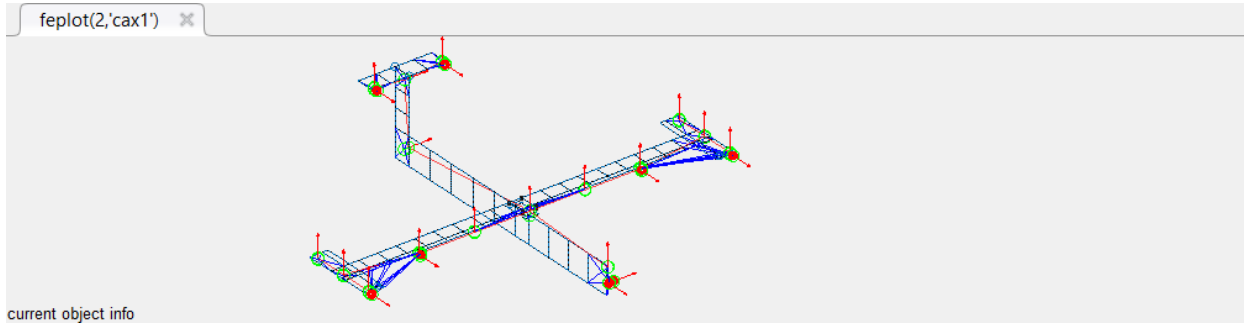
associated to basEst.



6. From this better relative position, one needs to iterate manually with small translations tx, ty, tz and rotations rx, ry, rz until the optimum is reached.



7. Finally, click on the button Accept associated to BasisToFEM to apply the coordinate transformation to the test wireframe and perform the compute the observation matrix of the FEM at sensors.



8. Clicking on `Finalize` will save the result in the corresponding project.

Another strategy using Iterative Closest Point algorithm is also implemented (in the `NodePairs` subtable). This will be documented in further release.

3.1.3 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 [14]) 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 `d_cor('TutoSensPlace')` demo. This function supports the effective independence [14] 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

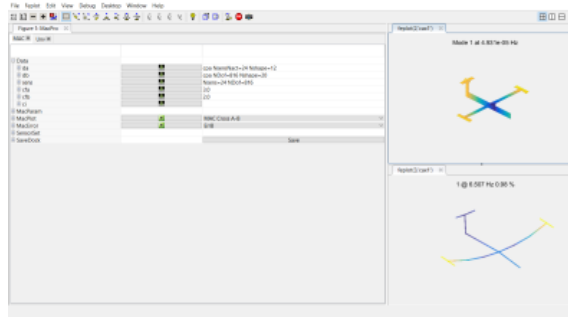
- you excite a set of target modes,
- or will have a combination of simultaneous loads that excites a particular mode and not other nearby modes.

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

3.2 Test/analysis correlation

Correlation criteria seek to analyze 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.



3.2.1 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) (theoretical description can be found in `ii_mac`). 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.

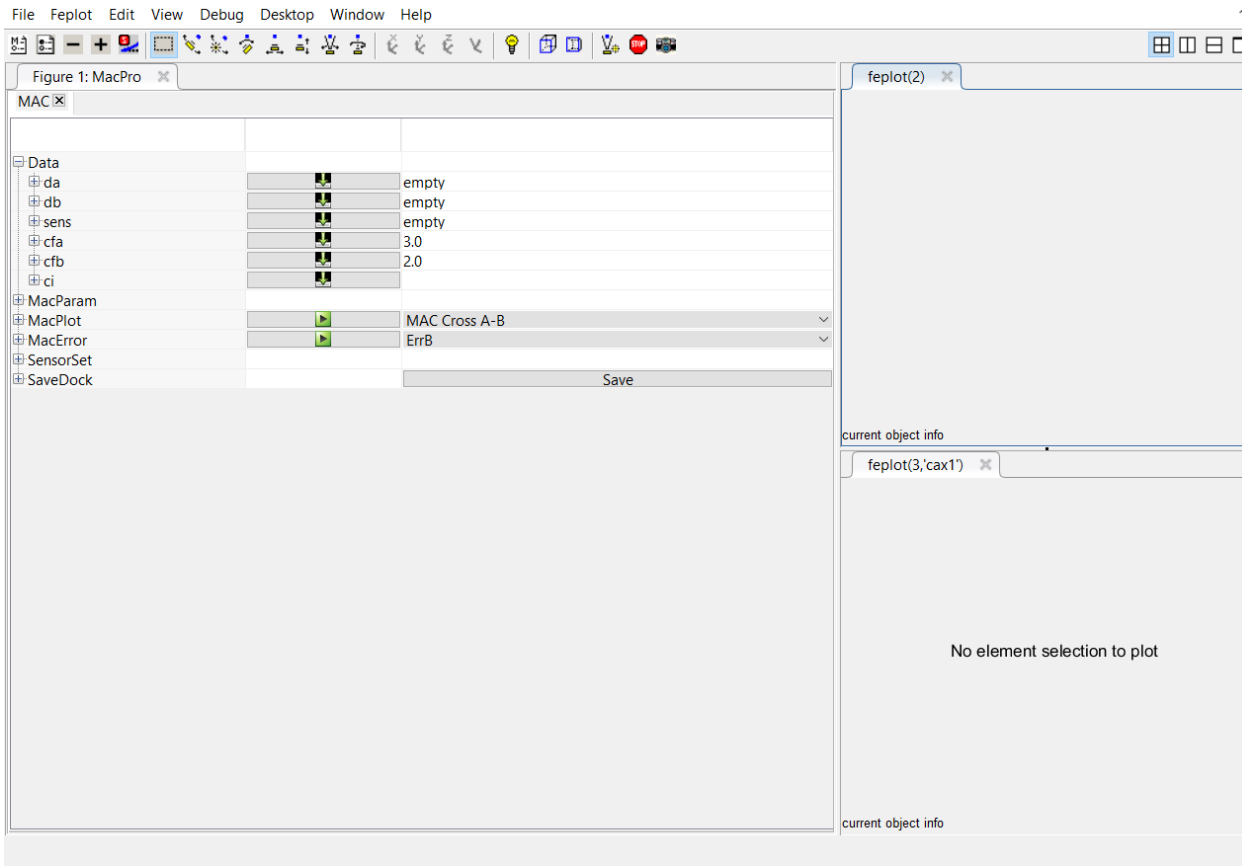
`MAC`	Modal Assurance Criterion (10.32). The most popular criterion for correlating vectors. Insensitive to vector scaling. Sensitive to sensor selection and level of response at each sensor. Main limitation : can give very misleading results without warning. Main advantage : can be used in all cases. A MAC criterion applied to frequency responses is called FRAC.
`POC`	Pseudo Orthogonality Checks (10.38). 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 (10.33) 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 (10.39). 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.

`ii_mac` describes the low-level calls to shape based correlation tools implemented in SDT, but to ease their practical usage, a dedicated `MAC` tab has been developed in the dock `CoShape`.

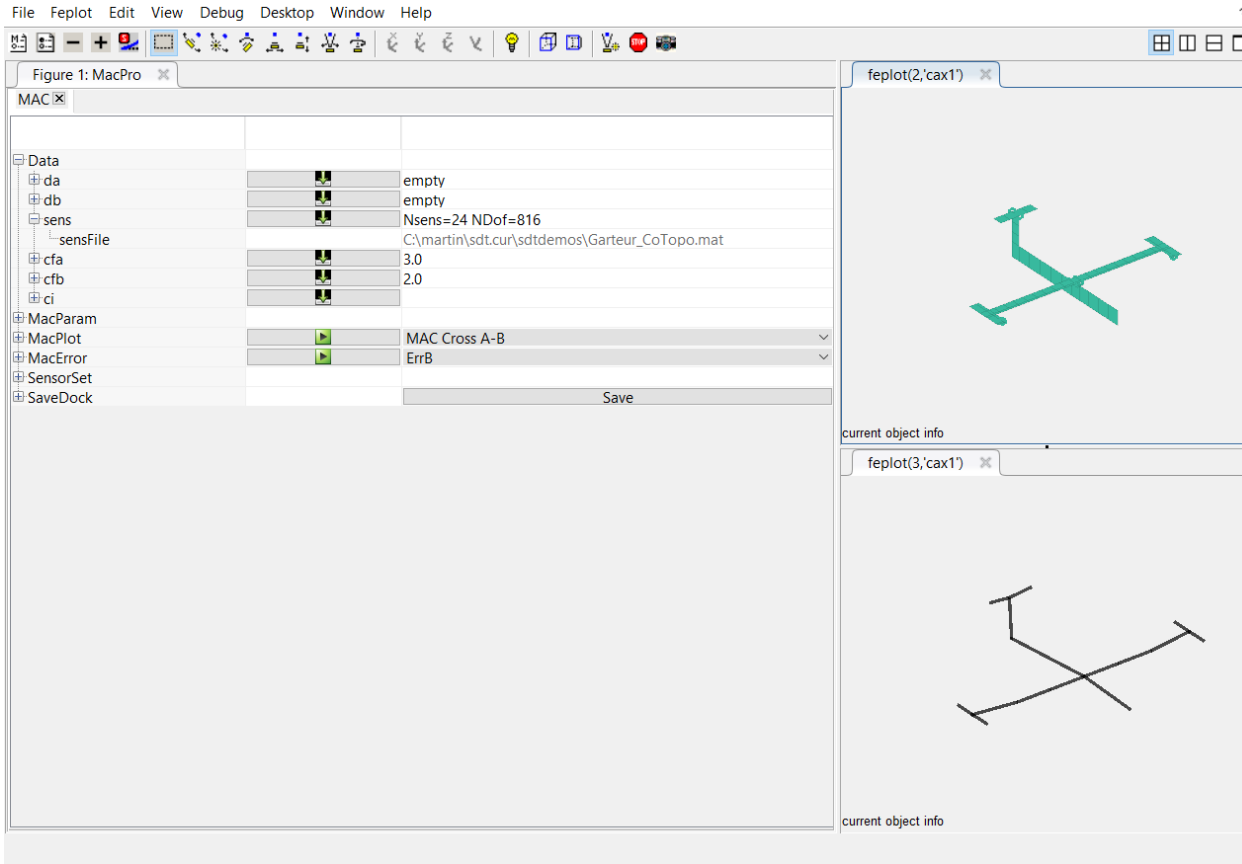
Here is a tutorial to present the classical GUI usage.

You will need the `garteur` example files, which can be found in *SDTPath/sdtdemos/gart*.m*. If these files are not present, click on the first step on the following tutorial in the HTML version of the documentation or download the patch at the address https://www.sdtools.com/contrib/garteur.zip and unzip the content in the folder *SDTPath/sdtdemos*.

1. Execute the command `fecom('dockCoShape')` to open an empty dock. You can also click on the button `CoShape` on the tree in `SDT Root`.



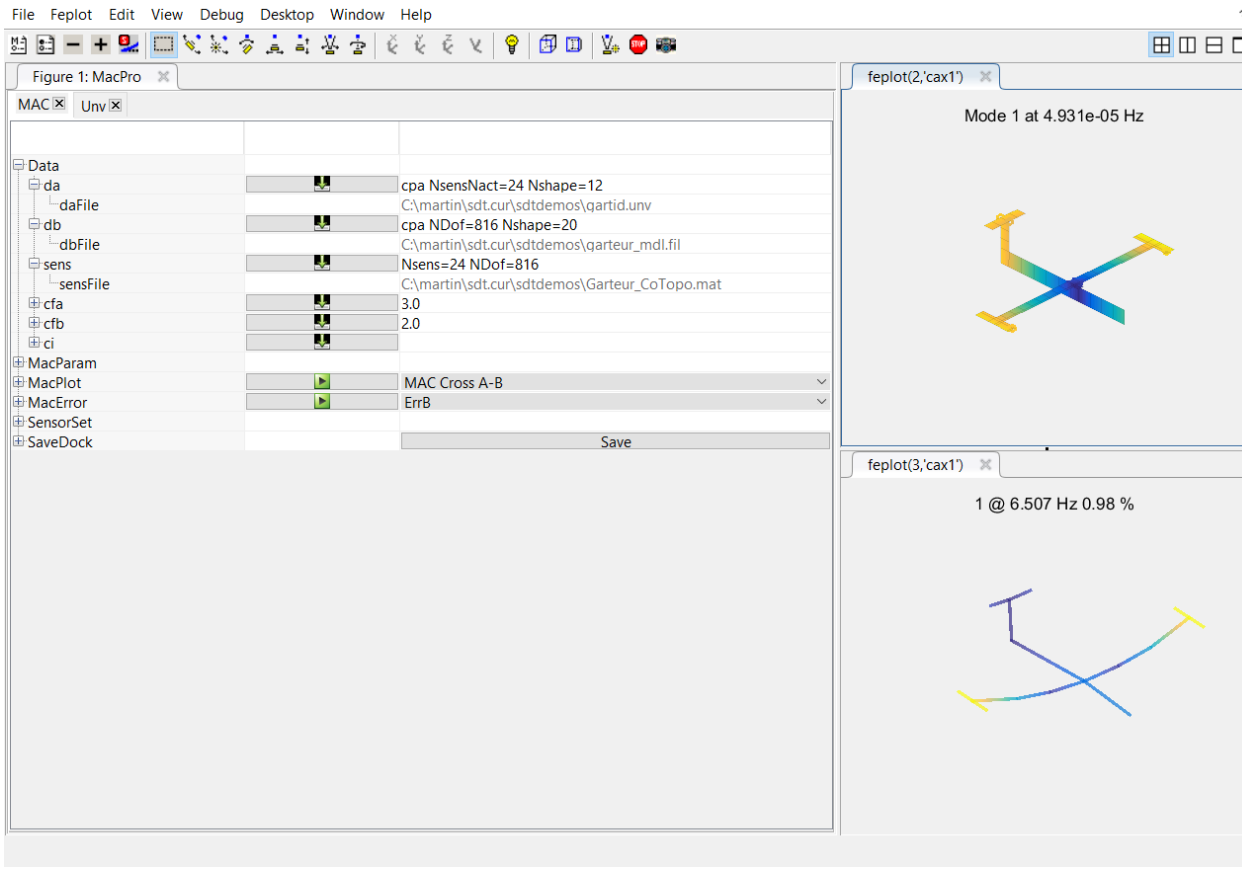
2. Click on associated to the line `sens` to open the file containing the result of the superposition between a test wireframe and a FEM. This will open the import model window. Select the file to load : for this tutorial, the file is located at `SDTPath/sdtdemos/gart_CoTopo.mat` (it corresponds to the dock `CoTopo` saved file of the tutorial in section 3.1.1). Data is loaded and displayed in the two `feplot` figures. A brief description of the number of the observation is given in the table providing the number of sensors `Nsens` and the number of FEM DOFs `NDof` for the observation matrix.



3. Click on associated to the line `da` to load the identified modes. In the opening window, select the file `SDTPath/sdtdemos/gartid.unv`. This will open the `Unv` tab in which you need to select the line containing the `shape data` and click on `import`.

   Do the same with the line `db` to load numerical modes. Select the file `SDTPath/sdtdemos/gart_mdl.fil` (mode computation result from abaqus).

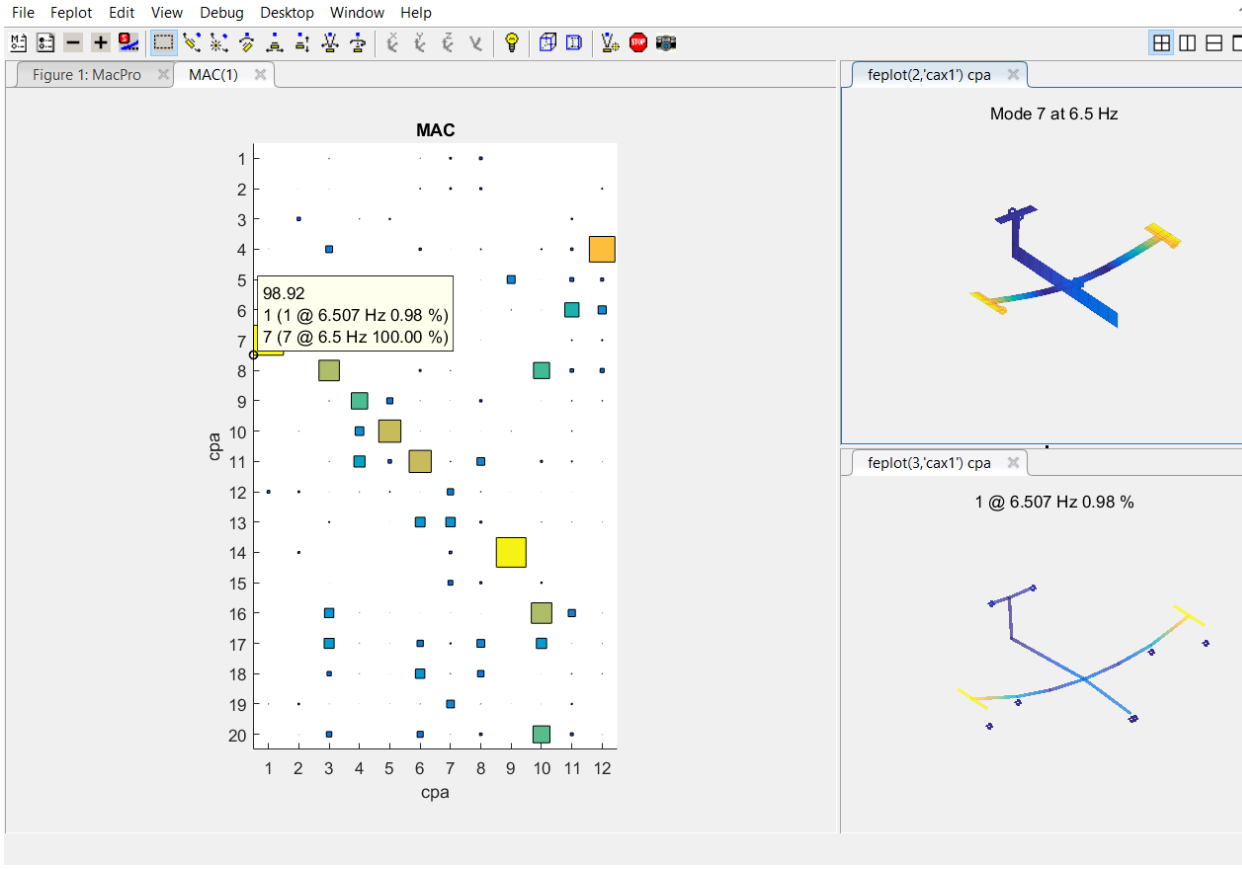
   The modeshapes are visible in both `feplot` figures.



A brief description the data is displayed:

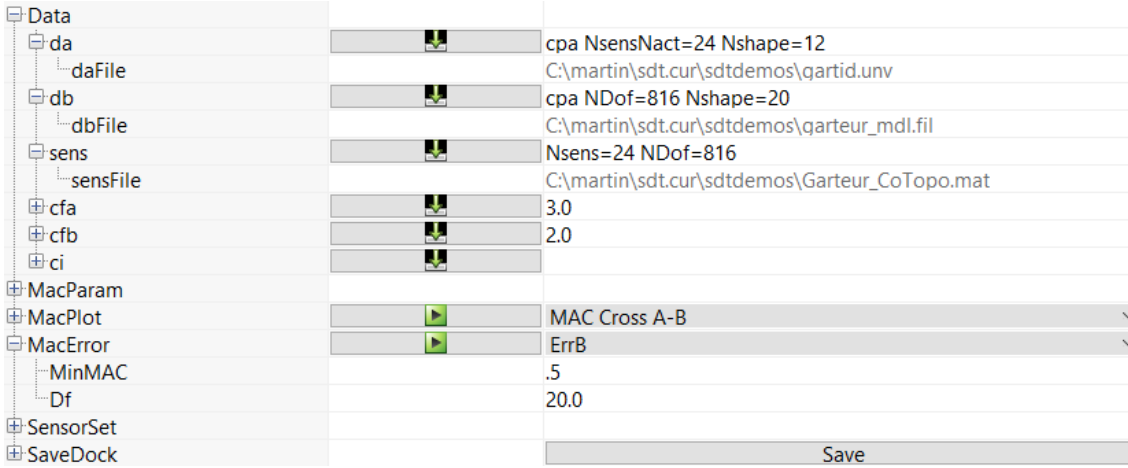
- for thee test `da`, the number of identified residues `NsensNact` and the number of shapes `Nshape`
- for thee FEM `db`, the number of DOF `Ndof` and the number of shapes `Nshape`

4. Click on associated to the line `MacPlot` to open the MAC matrix in a new window.

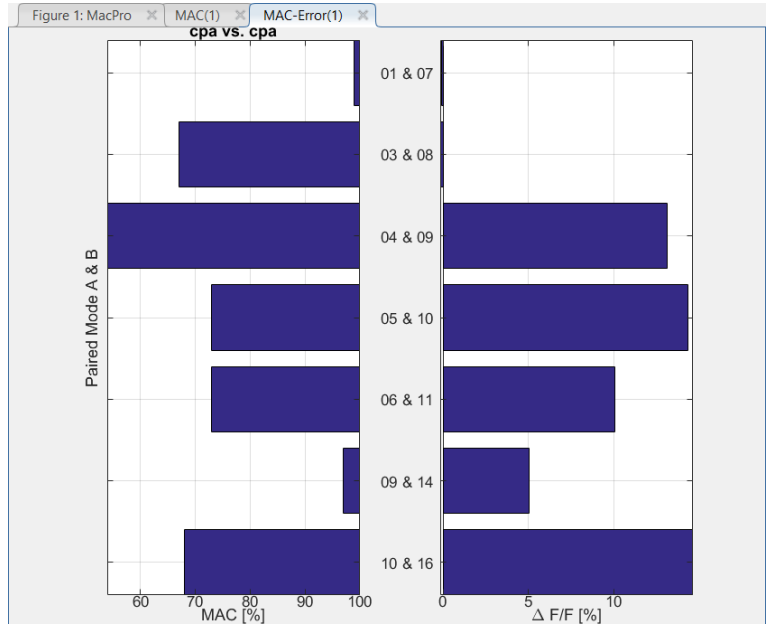


You can click on the square in the MAC matrix to interactively select the corresponding mode shapes in the `feplot` figure.

5. To pair more modes, expand the row `MacError` and allow a frequency shift `Df` of 20%.



Click then on associated to the line `MACError` to open the MACError display in a new window.



You can see here on the left the MAC value and on the right the relative frequency shift between the two sets of paired modes.

3.2.2 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 (6.95). Thus the dynamic residual

$$\left\{\hat{R}_j\right\} = \left[K - \omega_{\mathtt{jid}}^2 M\right] \{\phi_{\mathtt{id}j}\} \tag{3.1}$$

should be close to zero. A similar residual (3.5) 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

$$\{R_j\} = \left[\hat{K}\right]^{-1} \left[K - \omega_{\mathtt{jid}}^2 M\right] \{\phi_{\mathtt{id}j}\} \tag{3.2}$$

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 6.2.4), 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.

3.2.3 Correlation of FRFs

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

The quadratic cost function associated with the Euclidean norm

$$J_{ij}(\Omega) = \sum_{ij \text{ measured}, k \in \Omega} |\hat{H}_{ij}(s_k) - H_{ij}(s_k)|^2 \tag{3.3}$$

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

$$J_{ij}(\Omega) = \sum_{ij \text{ measured}, k \in \Omega} |log \left| \frac{\hat{H}_{ij}(s_k)}{H_{ij}(s_k)} \right| |^2 \tag{3.4}$$

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

$$\{R_j\} = \left[\hat{K}\right]^{-1} [[Z(\omega)] \{q_{ex}(\omega)\} - [b] \{u(\omega)\}] \tag{3.5}$$

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.

3.3 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 `d_cor` demonstration illustrates modeshape expansion in the *SDT*. This section summarizes the

theory and you are encouraged to download [15][16] from sdtools.com if you want more details.

3.3.1 Underlying theory for expansion methods

The unified perspective driving the *SDT* architecture is detailed in [15][16]. 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

$$\epsilon = \|\{y_{Test}\} - [c]\{q_{ex}\}\|_Q^2 \tag{3.6}$$

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 [17] 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 [18] is to compute the static response to the residual and use the associated strain energy, which is a good indicator of modeling error,

$$\|R_j(q_{ex})\|_K^2 = \{R_j\}^T \left[\hat{K}\right]^{-1} \{R_j\} \tag{3.7}$$

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 6.2.4). Variants of this energy norm of the dynamic residual can be found in [17].

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 3.3.2 . Standard subspace methods and their implementation are discussed in section section 3.3.3 . Methods taking modeling errors into account are discussed in section 3.3.4 .

3.3.2 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`. The easiest approach is typically a use of the `fe_sens WireExp` command as in the example below

```
% generate example, see sdtweb('demosdt.m#Sleeper')
cf=demosdt('sleeper');
TR=fe_sens('wireexp',cf.CStack{'Test'})
fe_sens('WireExpShow',cf,TR)
% display partial shapes as cell array
disp(TR)
r1=[{''} fe_c(TR.adof([1 3 5]))';
fe_def('subdof-cell',fe_def('subdef',TR,[1 3 5]),[1 2 46 48]')]
```

Given an interpolation matrix `TR`, you can animate interpolated shapes using `cf.def={def,exp}`. The interpolation (expansion) matrix `TR` has fields

- `TR.DOF` lists DOFs where the response is interpolated
- `TR.adof` lists input DOFs, these should match identifiers in the first column of a `sens.tdof` field.
- `TR.def` give the displacement at all DOFs corresponding to a unit sensor motion. Note as shown in the example above that a 1.08 $(1-y)$ measurement should lead to a negative value on the 1.02 $(1y)$ DOF. The same holds for measurements in arbitrary directions, `TR.def` should be unity when projected in the measurement direction.

The `fe_sens WireExp` command considers the wire frame as a coarse FEM model and uses expansion (see section 3.3.3 for details) to generate the interpolation. This is much more general than typical geometric constructions (linear interpolations, spline), which cannot handle arbitrary geometries.

Manual building of the interpolation matrix can be done by filling in the `TR.def` columns. `fe_sens('WireExpShow',cf,TR)` can then be used to verify the interpolation associated with each sensor (use the +/- buttons to scan trough sensors).

Starting from a basis of vectors `exp.def` with non unit displacements at the measurement DOFs, you can use

```
TR=exp;TR.adof=tdof(:,1);
TR.def=exp.def*pinv(fe_c(exp.DOF,tdof)*exp.def);
```

to minimize the norm of the test error (3.6) for a response within the subspace spanned by `exp.def` and thus generate a unmeasured DOF interpolation matrix.

3.3.3 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

$$\{q_R\} = \arg\min || \{y_{Test}\} - [c][T]\{q_R\} ||_2^2 \tag{3.8}$$

Modeshape expansion based on the subspace of low frequency modes is known as **modal** [19] or **SEREP** [20] 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 [15] 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 [21].

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 [22]. 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 [23] 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 6.2.4).

In practice, static expansion can be restated in the form (3.8) where T corresponds to constraint or modes associated to the load collocated to the output shape matrix characterizing sensors (see section 6.2). 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).

3.3.4 Model based expansion methods

Given metrics on test (3.6) and modeling (3.7) error, one uses a weighted sum of the two types of errors to introduce a generalized least-squares problem

$$min_{q_{j,ex}} \|R(q_{j,ex})\|_K^2 + \gamma_j \epsilon_j \tag{3.9}$$

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 [24]).

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.

3.4 Structural dynamic modification

While test results are typically used for test/analysis correlation and update, experimental data have 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 2.8.2) can be used to predict unmeasured

transfers. But these predictions lack residual terms (see section 6.2.3) 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).

```
ci=demosdt('demo gartid est');
ci.Stack{'Test'}.xf=-ci.Stack{'Test'}.xf;% driving 1012-z to 1012z
ci.Stack{'Test'}.dof(:,2)=12.03;
ci.IDopt.reci='1 FRF'; idcom(ci,'est');

ind=fe_c(ci.Stack{'IdMain'}.dof(:,1),[1012;1112],'ind');
po_ol=ci.Stack{'IdMain'}.po;

% Using normal modes
NOR = res2nor(ci.Stack{'IdMain'}); NOR.pb=NOR.cp';
S=nor2ss(NOR,'hz'); % since NOR.idopt tells acc. SS is force to Acc
mass=.3; a_cl = S.a - S.b(:,ind)*S.c(ind,:)*mass;
po_cln=ii_pof(eig(a_cl)/2/pi,3,2)
if sdtdef('UseControlToolbox-safe',1) && any(exist('ss','file')==[2 6]);
  SS=S;set(SS,'b',S.b(:,4),'d',S.d(:,4),'InputName',S.InputName(4))
else % Without CTbox
  SS=S;SS.b=SS.b(:,4);SS.d=SS.d(:,4);SS.dof_out=SS.dof_out(4,:);
end
qbode(SS,ci.Stack{'Test'}.w*2*pi,'iiplot "Normal"');

% Using complex modes
SA = res2ss(ci.Stack{'IdMain'},'AllIO');
a_cl = S.a - S.b(:,ind)*S.c(ind,:)*mass;
po_clx=ii_pof(eig(a_cl)/2/pi,3,2)
if sdtdef('UseControlToolbox-safe',1) && any(exist('ss','file')==[2 6]);
  SS=SA;set(SS,'b',S.b(:,4),'d',S.d(:,4)*0,'InputName',S.InputName(4))
else % Without CTbox
```

```
  SS=SA;SS.b=SS.b(:,4);SS.d=SS.d(:,4)*0;SS.dof_out=S.dof_out(4,:);
end
qbode(SS,ci.Stack{'Test'}.w*2*pi,'iiplot "Cpx"');
iicom('ch4');

% Frequencies
figure(1);in1=1:8;subplot(211);
bar([ po_clx(in1,1) po_cln(in1,1)]./po_ol(in1,[1 1]))
ylabel('\Delta F / F');legend('Complex modes','Normal modes')
set(gca,'ylim',[.5 1])

% Damping
subplot(212);bar([ po_clx(in1,2) po_cln(in1,2)]./po_ol(in1,[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 `ci.Stack{'Test'}.xf` 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 [25] or get in touch with SDTools for our latest advances on the subject.

FEM tutorial

4.1 FE mesh declaration 139
4.1.1 Direct declaration of geometry (truss example) 139
4.2 Building models with feutil 140
4.3 Building models with femesh 144
4.3.1 Automated meshing capabilities 146
4.3.2 Importing models from other codes 146
4.3.3 Importing model matrices from other codes 147
4.4 The feplot interface 149
4.4.1 The main feplot figure 149
4.4.2 Viewing stack entries 153
4.4.3 Pointers to the figure and the model 153
4.4.4 The property figure 153
4.4.5 GUI based mesh editing 155
4.4.6 Viewing shapes 156
4.4.7 Viewing property colors 158
4.4.8 Viewing colors at nodes 159
4.4.9 Viewing colors at elements 159
4.4.10 feplot FAQ 160
4.5 Other information needed to specify a problem 162
4.5.1 Material and element properties 162
4.5.2 Other information stored in the stack 164
4.5.3 Cases GUI 164
4.5.4 Boundary conditions and constraints 166
4.5.5 Loads 167
4.6 Sensors 168
4.6.1 Sensor GUI, a simple example 168
4.6.2 Sensor definition from a cell array 169

4.6.3 Sensor data structure and init commands 173
4.6.4 Topology correlation and observation matrix 180
4.7 Stress observation 184
4.7.1 Building view mesh 184
4.7.2 Building and using a selection for stress observation 186
4.7.3 Observing resultant fields 187
4.8 Computing/post-processing the response 187
4.8.1 Simulate GUI 187
4.8.2 Static responses 188
4.8.3 Normal modes (partial eigenvalue solution) 189
4.8.4 State space and other modal models 190
4.8.5 Viewing shapes, stress, energy, ... 192
4.8.6 Time computation 192
4.8.7 Manipulating large finite element models 194
4.8.8 Optimized assembly strategies 196

This chapter introduces notions needed to use finite element modeling in the *SDT*. It illustrates how to define mechanical problems (model, boundary conditions, loads, etc.), compute and post-process the response

- using the `feplot` Graphical User Interface,
- or using script commands.

The GUIs are described and the connections between graphical and low level data are detailed for

- the model data structures,
- the case (i.e. DOFs, boundary conditions, loads, ...),
- the response to a specified case,
- the results post-processing .

4.1 FE mesh declaration

This section gives a summary of FE mesh declaration with pointers to more detailed documentation.

4.1.1 Direct declaration of geometry (truss example)

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

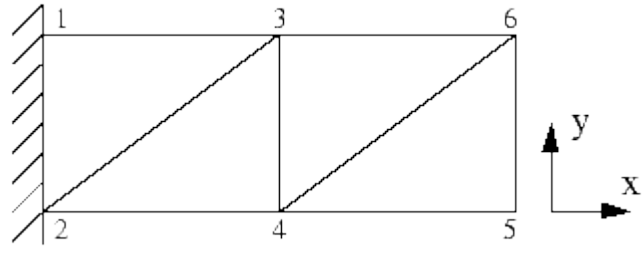


Figure 4.1: FE model.

In `d_mesh('TutoBmesh-s1')` , the geometry is declared in the `model.Node` matrix (see section 7.1 and section 7.1.1). In this case, one defines 6 nodes for the truss and an arbitrary reference node to distinguish principal bending axes (see `beam1`)

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

The model description matrix (see section 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 ];
```

4.2 Building models with feutil

Declaration by hand is clearly not the best way to proceed in general.`feutil` provides a number of commands for finite element model creation.`feutil` should be preferred to `femesh` which is a lower level command. One can find meshing examples through the `feutil` commands in

- `d_truss` : this demo builds a truss model using beam elements.
- `d_ubeam` : the beginning of the demo builds a volume model that is used is various examples of this documentation.

The principle of feutil meshing strategy is to build sub model parts using the feutil basic meshing commands (extrusion, rotation, revolution, division, ...) and to assemble those models to form the resulting model thanks to the `feutil AddTest` commands.

Following detailed example builds the GARTEUR model.

First the model data structure is initialized (see `sdtweb model`), with fields `Node` (that contains some initial nodes that will be used to begin building of elements by elementary operations), `Elt` (which is empty at this step), `unit` (that contains the unit of the mesh, that must be coherent with material properties defined later. Here the SI system is used that means that node positions are defined in meters.), and `name` (that contains model name that is used to identify the model in the assembly steps for example).

```
%% Step1 : Initialize model
model=struct('Node',[1 0 0 0  0   0   0;     2 0 0 0 0   0   0.15;
                     3 0 0 0  0.4 1.0 0.176; 4 0 0 0 0.4 0.9 0.176],...
             'Elt',[],'unit','SI','name','GARTEUR');
```

Now the fuselage is built by creating an initial beam between nodes 1 and 2 (see `feutil Object` commands to easily create a number of elementary models). Then the beam is extruded with an irregular spatial step in the x direction, to form `quad4` elements that represents the fuselage.

```
%% Step2 Fuselage
model.Elt=feutil('ObjectBeamLine 1 2',model);
model=feutil('Extrude 0  1.0 0.0 0.0',model,...
             [linspace(0,.55,5) linspace(.65,1.4,6) 1.5]);
```

The same strategy is used to mesh the quads corresponding to the plane tail. The extremities of the initial beam to be extruded are not explicitely defined as previously, but are found in the nodes created in the last step through the `feutil FindNode` command (that returns the NodeId of nodes found by `FindNode`). Here nodes are found at z position equal to .15, and x upper than 1.4. The vertical tail is built in a temporary model named `mo0`. Note that `mo0` is first initialized with principal `model` nodes (`mo0=model;`) so that new nodes that will be added during the extrusion respect the NodeId numerotation of the main model. Then we can simply add the vertical tail `mo0` to the main model using the `feutil AddTestCombine` command (if node numerotation was not coherent for the new part `mo0` and the main `model` already defined nodes, we would have to use the `feutil AddTestMerge` command that can be really time consuming).

```
%% Step3 vertical tail
n1=feutil('FindNode z==.15 & x>=1.4',model);
mo0=model; mo0.Elt=feutil('ObjectBeamLine',n1);
mo0=feutil('Extrude 3 0 0 .1',mo0);
model=feutil('AddTestCombine-noori',model,mo0);
```

Then the vertical horizontal tail, the right and left drums, the wings and the connection plate are built and added to main `model` using the same strategy:

```
%% Step4 Vertical horizontal tail
n1=feutil('FindNode z==.45',model)
mo0=model; mo0.Elt=feutil('ObjectBeamLine',n1);
mo0=feutil('Extrude 0  0.0 0.2 0.0',mo0,[-1 -.5 0 .5 1]);
model=feutil('AddTestCombine;-noori',model,mo0);

%% right drum
mo0=model; mo0.Elt=feutil('ObjectBeamLine 3 4');
mo0=feutil('Extrude 1 .4 0 0',mo0);
mo0=feutil('Divide',mo0,[0 2/40 15/40 25/40 1],[0 .7 1]);
model=feutil('AddTestCombine;-noori',model,mo0);

%% left drum
mo0=feutil('SymSel 1 0 1 0',mo0);
model=feutil('AddTestCombine;-noori',model,mo0);

%% wing
n1=feutil('FindNode y==1 & x>=.55 & x<=.65',model);
mo0=model; mo0.Elt=feutil('ObjectBeamLine',n1);
mo0=feutil('Divide',mo0,[0 1-.762 1]);
mo0=feutil('Extrude 0  0.0 -1.0 0.0',mo0,[0 0.1 linspace(.15,.965,9) ...
                                          linspace(1.035,1.85,9) 1.9 2.0]);
model=feutil('AddTestCombine;-noori',model,mo0);

%% Connection plate
n1=feutil('FindNode y==0.035 | y==-0.035 & x==.55',model)
mo0=model; mo0.Elt=feutil('ObjectBeamLine',n1);
mo0=feutil('Divide 2',mo0);
mo0=feutil('TransSel -.02 0 0',mo0);
mo0=feutil('Extrude 0 1 0 0',mo0,[0 .02 .12 .14]);
i1=intersect(feutil('FindNode group6',model),feutil('FindNode group1',mo0));
mo0=feutil('TransSel 0.0 0.0 -0.026',mo0);
model=feutil('AddTestCombine;-noori',model,mo0);
```

The stiffness connecting the connection plate are built extruding a mass object to form a beam, and then changing the name of the beam group as `celas` which are the spring elements in SDT.

```
%% Step5 Stiff links for the connection
```

```
mo0=model; mo0.Elt=feutil('Object mass',i1);
mo0=feutil('Extrude 1 0 0 -.026',mo0);
mo0.Elt=feutil('set group1 name celas',mo0);
```

The celas properties are defined in the element matrix (see `sdtweb celas` for more details). First row of `mo0` is the header, the springs are stored as following rows (2nd row to the end). The springs connect the master DOF (column 3) x, y, z, θ_x and θ_y to the same DOF on the slave nodes (column 4, 0 that mean the same as master). The stiffness (column 7) is defined at 1e12. The 4 springs in`mo0` are then added to the main `model`.

```
%% Step6 set connected DOFs and spring value
mo0.Elt(2:end,3)=12345; % master dof
mo0.Elt(2:end,4)=0; % same dof as master
mo0.Elt(2:end,7)=1e12; % stiffness
model=feutil('AddTestCombine;-noori',model,mo0); % add springs to main model
```

Then group 6 is divided in 2 groups to get the part covered by constraining layer in a separated group (in order to help the later manipulations of this part, such as material identifier definition).

```
%% Step7 Make a group of the part covered by the constraining layer
model.Elt=feutil('Divide group 6 InNode {x>.55 & y<=.85 & y>=-.85}',model);
```

Then some masses are added through the `ObjectMass` command. Then all masses are regrouped in a same group.

```
%% Step8 Tip masses
i1=feutil('FindNode y==0.93 | y==-0.93 & x==0.42',model)
mo0=model; mo0.Elt=feutil('Object mass',i1,[0.2 0.2 0.2]); %200g
model=feutil('AddTestCombine;-noori',model,mo0);
i1=feutil('FindNode z==.45 & y==0',model)
mo0=model; mo0.Elt=feutil('Object mass',i1,[0.5 0.5 0.5]); %500g
model=feutil('AddTestCombine;-noori',model,mo0);
model=feutil('Join mass1',model); % all mass in the same group
```

Then plates are oriented (see the `feutil Orient` command) so that offset in correct direction can be defined. Offset (distances in the normal direction from element plane to reference plane) are defined in element matrices in the 9th column for `quad4` elements. The `feutil FindElt` command is used to find the indices of considered elements in the model element matrix `model.Elt`.

```
%% Step9 Orient plates that will need an off-set
model.Elt=feutil('Orient 4:8 n 0 0 3',model);
i1=feutil('FindElt group4:5',model);
model.Elt(i1,9)=0.005; % drums (positive off-set)
i1=feutil('FindElt group6:7',model);
```

```
model.Elt(i1,9)=-0.005; % wing
i1=feutil('FindElt group8',model);
model.Elt(i1,9)=0.008; % wing
```

Now `ProId` (element property identifier) and `MatId` (material identifier) are defined for each element. In last meshing steps, elements have been added by group (or separated), so that we only attribute a material and element property identifier for each group.

```
%% Step10 Deal with material and element properties identifier:
model.Elt=feutil('Set group1 mat1 pro3',model);
model.Elt=feutil('Set group2:7 mat1 pro1',model);
model.Elt=feutil('Set group8 mat2 pro2',model);
model.Elt=feutil('Set group6 pro4',model);
```

And following lines define associated properties:

```
%% Step11 Define associated properties:
model.pl=[m_elastic('dbval 1 aluminum');
          m_elastic('dbval 2 steel')];
model.il = [1 fe_mat('p_shell','SI',1)  2 1 0     .01
            2 fe_mat('p_shell','SI',1)  2 1 0     .016
            3 fe_mat('p_shell','SI',1)  2 1 0     .05
            4 fe_mat('p_shell','SI',1)  2 1 0     .011];
```

The result is then displayed in feplot, coloring each material differently:

```
%% Step12 Display in feplot
 cf=comgui('guifeplot -project "SDT Root"',3); % Robust open in figure(3)
 cf.model=model;       % display model
 fecom(';sub 1 1;view3; colordatamat-edgealpha.1'); % 1 subplot, specify view, color,
```

4.3 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

```
%% Step1 Declare nodes and build single bay
 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

```
%% Step2 Put in main model, translate seclection and add to 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)

```
%% Step3 Create a 3D struss based on a single 2D bay
 femesh('reset');
 femesh('test2bay');
 femesh('removeelt group2');
 femesh('divide group 1 InNode 1 4');
 femesh('set group1 name bar1');
 femesh(';selgroup2 1;repeatsel 10 1 0 0;addsel');
 femesh(';rotatesel 1 60 1 0 0;addsel;');
 femesh(';selgroup3:4;rotatesel 2 -60 1 0 0;addsel;');
 femesh(';selgroup3:8');
 model=femesh('model0');  % export FEnode and FEel0 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 tutorial scripts listed using d_mesh('Tuto') or in scripts beambar, d_ubeam, gartfe.

4.3.1 Automated meshing capabilities

While this is not the toolbox focus, SDT supports some free meshing capabilities.

fe_gmsh is an interface to the open source 3D mesher GMSH. Calls to this external program can be used to generate meshes by direct calls from MATLAB. Examples are given in the function reference.

fe_tetgen is an interface to the open source 3D tetrahedral mesh generator. See help fe_tetgen for commands.

fe_fmesh('qmesh') implements a 2D quad mesher which meshes a coarse mesh containing triangles or quads into quads of a target size. All nodes existing in the rough mesh are preserved. The -noTest option removes the initial mesh.

```
% build rough mesh
model=feutil('Objectquad 1 1',[0 0 0;2 0 0; 2 3 0; 0 3 0],1,1);
model=feutil('Objectquad 1 1',model,[2 0 0;8 0 0; 8 1 0; 2 1 0],1,1);
% start the mesher with characteristic length of .1
model=fe_fmesh('qmesh .1',model.Node,model.Elt);
feplot(model);
```

Other resources in the MATLAB environment are initmesh from the PDE toolbox and the Mesh2D package.

4.3.2 Importing models from other codes

The base *SDT* supports reading/writing of test related Universal 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.

To open the FEMLink GUI use sdtroot('InitFEMLink'). for a reference on the FEMLink Tab, see section 8.2.2 . 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 4.3.3 .

These interfaces evolve with user needs. Please don't hesitate to ask for a patch even during an SDT evaluation by sending a test case to info@sdtools.com.

Interfaces available when this manual was revised were

ans2sdt	reads ANSYS binary files, reads and writes .cdb input (see FEMLink)
abaqus	reads ABAQUS binary output .fil files, reads and writes input and matrix files (.inp,.mtx) (see FEMLink)
nasread	reads the MSC/NASTRAN [26] .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.
nopo	This OpenFEM function reads MODULEF models in binary format.
perm2sdt	reads PERMAS ASCII files (this function is part of FEMLink)
samcef	reads SAMCEF text input and binary output .u18, .u11 , .u12 files (see 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.

4.3.3 Importing model matrices from other codes

FEMLink handles importing element matrices for NASTRAN (nasread BuildUp), ANSYS (ans2sdt Build), SAMCEF (samcef read) and ABAQUS (abaqus read).

Reading of full matrices is supported for NASTRAN in the binary .op2 and .op4 formats (writing to .op4 is also available). For ANSYS, reading of .matrix ASCII format is supported. For ABAQUS, reading of ASCII .mtx format is supported.

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

To incorporate full FEM matrices in a SDT model, 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. If you have no mesh, define nodes and associated mass elements.

fesuper provides functions to handle superelements. In particular, fesuper SEAdd lets you define a superelement model, without explicitly defining nodes or elements (you can specify only DOFs and element matrices), and add it to another model.
Following example loads ubeam model, defines additional stiffness and mass matrices (that could have been imported) and a visualization mesh.

```
% Load ubeam model :
model=demosdt('demo ubeam-pro');
cf=feplot; model=cf.mdl;
% Define superelement from element matrices :
SE=struct('DOF',[180.01 189.01]',...
    'K',{{[.1 0; 0 0.1] 4e10*[1 -1; -1 1]}},...
    'Klab',{{'m','k'}},...
    'Opt',[1 0;2 1]); % Matrix types, sdtweb secms#SeStruct
% Define visualization mesh :
SE.Node=feutil('GetNode 180 | 189',model);
SE.Elt=feutil('ObjectBeamLine 180 189 -egid -1');
% Add as a superelement to model :
model=fesuper('SEadd -unique 1 1 selt',model,SE);
```

You can easily define weighting coefficient associated to matrices of the superelement, by defining an element property (see p_super for more details). Following line defines a weighting coefficient of 1 for mass and 2 for stiffness (1001 is the MatId of the superelement).

```
% Define weighting coefficients for mass and stiffness matrices
model.il=[1001 fe_mat('p_super','SI',1) 1 2];
```

You may also want to repeat the superelement defined by element matrices. Following example shows how to define a model, from repeated superelement:

```
% Define matrices (can be imported from other codes) :
model=femesh('testhexa8');
[m,k,mdof]=fe_mk(model);
% Define the superelement:
SE=struct('DOF',[180.01 189.01]',...
    'K',{{[.1 0; 0 0.1] 4e10*[1 -1; -1 1]}},...
    'Klab',{{'m','k'}},...
    'Opt',[1 0;2 1]);
```

```
SE.Node=model.Node; SE.Elt=model.Elt;
% Add as repeated superelement:
% (need good order of nodes for nodeshift)
model=fesuper('SEAdd -trans 10 0.0 0.0 1.0 4 1000 1000 cube',[],SE);
cf=feplot(model)
```

Superelement based substructuring is demonstrated in d_cms2 which gives you a working example where model matrices are stored in a generic superelement. Note that numerical precision is very important when importing model matrices. Storing matrices in 8 digit ASCII format is very often not sufficient.

4.4 The feplot interface

Three kinds of manipulations are possible using the feplot GUI

- viewing the model and post-processing the responses,
- setting and displaying the mechanical problem (model properties and cases),
- setting the view properties.

4.4.1 The main feplot figure

feplot figures are used to view FE models and hold all the data needed to run simulations. Data in the model can be viewed in the property figure (see section 4.4.4). Data in the figure can be accessed from the command line through pointers as detailed in section 4.4.3 . The feplot help gives architecture information, while fecomlists available commands. Most demonstrations linked to finite element modeling (see section 1.1 for a list) give examples of how to use feplot and fecom.

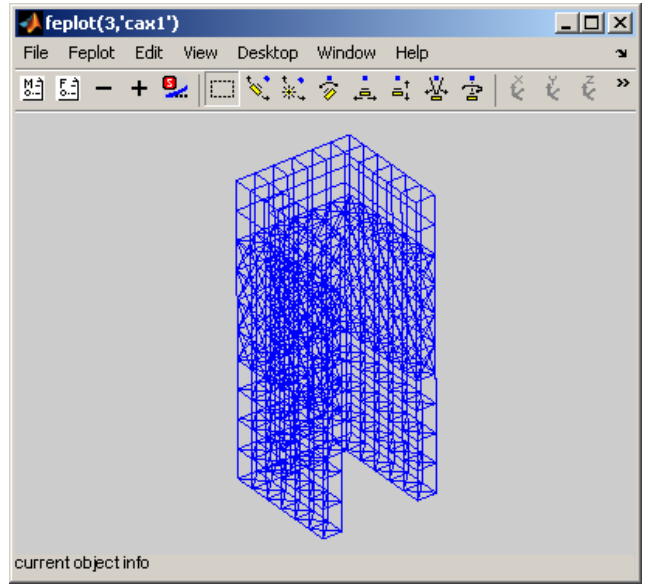


Figure 4.2: Main feplot figure.

The first step of most analyzes is to display a model in the main `feplot` figure. Examples of possible commands are (see `fecom load` for more details)

- `cf=feplot(model)` display the model in a variable and returns a pointer object `cf` to the figure.
- `cf=feplot(5);cf.model=model;` do the same thing but in figure 5.
  `cf=feplot;cf.model={node,elt};` will work for just nodes and elements. Note that `cf.model` is a method to define the model and is not a pointer. `cf.mdl` is a pointer to the model, see section 4.4.3 .
- `feplot('load','File.mat')` load a model from a `.mat` file.

As an example, you can 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

```
model=demosdt('demogartfe')
cf=feplot;       % open FEPLOT and define a pointer CF to the figure
cf.model=model;
```

The main capabilities the `feplot` figure are accessible using the figure toolbar, the keyboard shortcuts, the right mouse button (to open context menus) and the menus.

Toolbar

List of icons used in GUIs

	Model properties used to edit the properties of your model.
	Start/stop animation
	Previous Channel/Deformation
	Next Channel/Deformation
	`iimouse zoom`
	Orbit. Remaining icons are part of MATLAB `cameratoolbar` functionality.
	Snapshot. See `iicom ImWrite`.

Keyboard shortcuts

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 any of the `u`, `U`, `v`, `V`, `w`, `W`, `3`, `2` keys to rotate the plot (press the `?` key for a list of `iimouse`key shortcuts).

Menus and context menu

The `contextmenu` associated with your plot may be opened 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` (associated with the figure, the axes and objects). A few important functionalities and the associated commands are

- `Cursor Node` tracks mouse movements and displays information about pointed object. This is equivalent to the `iimouse('cursor')` command line.
- `Cursor...[Elt,Sel,Off]` selects what information to display when tracking the mouse. The `iimouse('cursor[onElt,onSel,Off]')` command lines are possible.
- `Cursor... 3DLinePick` (which can be started with `fe_fmesh('3DLineInit')`) allows node picking. Once started, the context menu gives access `info` (lists picked nodes and distances) and `done` prints the list of picked nodes.
- `TextNode` activates the node labeling. It is equivalent to the `fecom('TextNode')` command line.

- `Triax` displays the orientation triax. It is equivalent to the `fecom('triax')` command line.
- `Undef` shows the undeformed structure. Other options are accessible with the `fecom('undef[dot,line]')` command line.
- `Views...` [View n+x,...] selects default plot orientation. The `iimouse('[vn+x,...]')` command lines are available.
- `colorbar on` shows the colorbar, for more accurate control see `fecom ColorBar`.
- `Zoom Reset` is the same as the `iimouse('resetvie')` command line to reset the zoom.
- `setlines` is the same as the `setlines` command line.

The figure `Feplot` menu gives you access to the following commands (accessible by `fecom`)

- `Feplot:Feplot/Model properties` opens the property figure (see section 4.4.4).
- `Feplot:Sub commands:Sub IsoViews` (same as `iicom('subiso')`) gets a plot with four views of the same mode. Use `iicom('sub2 2 step')` to get four views of different modes.
- `Feplot:Show` menu generates standard plots. For FE analyses one will generally use surface plots color-coded surface plots using `patch` objects) or wire-frame plots (use `Feplot:Show` menu to switch).
- `Feplot:Misc` shows a `Triax` or opens the channel selector.
- `Feplot:Undef` is used to show or not the undeformed structure.
- `Feplot:Colordata` shows structure with standard colors.
- `Feplot:Selection` shows available selections.
- `Feplot:Renderer` is used to choose the graphical rendering. Continuous animation in OpenGL rendering is possible for models that are not too large. The `fecom SelReduce` can be use to coarsen the mesh otherwise.
- `Feplot:Anim` chooses the animation mode.
- `Feplot:View defaults` changes the orientation view.

4.4.2 Viewing stack entries

You can typically view stack entries by clicking on the associated entry and using ProViewOn (icon). Handling of which deformation is shown in multi-channel entries is illustrated below

```
model=demosdt('demo UbeamDofLoad');cf=feplot;
fecom('curtabCases','Point load 1');fecom('proViewOn');

% Control channel in multi column DOFLoad
cf.CStack{'Point load 1'}.Sel.ch=2;fecom('proViewOn');
```

4.4.3 Pointers to the figure and the model

`cf1=feplot` returns a pointer to the current `feplot` figure. 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)`. If many `feplot` figures are open, one can define the target giving an `feplot` figure handle `cf` as a first argument to `fecom` commands.

The model is stored in a graphical object. `cf.model` is a method that calls `fecom InitModel`. `cf1.mdl` is a method that returns a pointer to the model. Modifications to the pointer are reflected to the data stored in the figure. However `mo1=cf.mdl;mo1=model` makes a copy of the variable `model` into a new variable `mo1`.

`cf.Stack` gives access to the model stack as would `cf.mdl.Stack` but allows text based access. Thus `cf.Stack{'EigOpt'}` searches for a name with that entry and returns an empty matrix if it does not exist. If the entry may not exist a type must be given, for example `cf.Stack{'info','EigOpt'}=[5 10 1]`.

`cf.CStack` gives access to the case stack as would calls of the form

`Case=fe_case(cf.mdl,'getcase');stack_get(Case,'FixDof','base')` but it allows more convenient string based selection of the entries.

`cf.Stack` and `cf.CStack` allow regular expressions text based access. First character of such a text is then `#`. One can for example access to all of the stack entries beginning by the string `test` with `cf.Stack{'#test.*'}`. Regular expressions used by `SDT` are standard regular expressions of `Matlab`. For example `.` replaces any character, `*` indicates 0 to any number repetitions of previous character...

4.4.4 The property figure

Finite element models are described by a data structures with the following main 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, etc.), material names, etc.

The model content can be viewed using the `feplot` property figure. This figure is opened using the icon, or `fecom('ProInit')`.

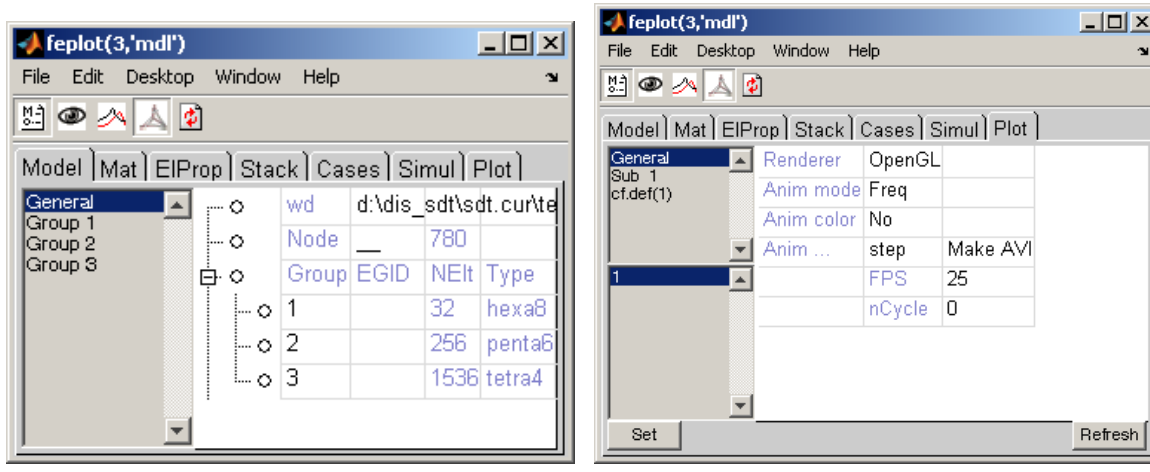


Figure 4.3: Model property interface.

This figure has the following tabs

- `Model` tab gives general information on the model nodes and elements. You can declare those by hand as shown in section 4.1.1 , through structured mesh manipulations with `feutil` see section 4.3 , or through import see section 4.3.2 . (see section 4.5 and Figure 4.3). You can visualize one or more groups by selecting them in the left group list of this tab.
- `Mat` tab lists and edits all the material. In the mode, associated elements in selection are shown. See section 4.5.1 .
- `ElProp` tab lists and edits all the properties. See section 4.5.1 .
- `Stack` tab lists and edits general information stored in the model (see section 7.7 for possible

entries). You can access the model stack with the `cf.Stack` method.

- `Cases` tab lists and edits load and boundary conditions (see section 4.5.3 and Figure 4.9). You can access the case stack with the `cf.CStack` method.
- `Simulate` tab allows to launch the static and dynamic simulation (see section 4.8 and Figure 4.12).

The figure icons have the following uses

	Model properties used to edit the properties of your model.
	Active display of current group, material, element property, stack or case entry. Activate with `fecom('ProViewOn');`
	Open the iiplot GUI.
	Open/close feplot figure
	Refresh the display, when the model has been modified from script.

4.4.5 GUI based mesh editing

This section describes functionality accessible with the `Edit` list item in the `Model` tab. To force display use `fecom('CurtabModel','Edit')`.

- `AddNode` opens a dialog that lets you enter nodes by giving their coordinates `x y z`, their node number and coordinates `NodeId x y z` or all the node information `NodeId CID DID GID x y z`.
- `AddNodeCG` starts the 3D line picker. You can then select a group of nodes by clicking with the left button on these nodes. When you select `Done` with the context menu (right click), a new node is added at the CG of the selected nodes.
- `AddNodeOnEdge` starts the 3D line picker to pick two nodes and adds nodes at the middle point of the segment.
- `AddElt Name` starts the 3D line picker and lets you select nodes to mesh individual elements. With `Done` the elements are added to the model as a group.
- `AddRbe3` starts a line picker to define an RBE3 constraint. The first node picked is slave to the motion of other nodes.
- `RemoveWithNode` starts the 3D line picker. You can then select a group of nodes by clicking with the left button on these nodes. When you select `Done` with the context menu (right click), elements containing the selected nodes are removed.

- RemoveGroup opens a dialog to remove some groups.

Below are sample commands to run the functionality from the command line.

```
model=demosdt('demoubeam');cf=feplot;
fecom('CurtabModel','Edit')
fecom(cf,'addnode')
fecom(cf,'addnodecg')
fecom(cf,'addnodeOnEdge')
fecom(cf,'RemoveWithNode')
fecom(cf,'RemoveGroup')
fecom(cf,'addElt tria3')

fe_case(cf.mdl,'rbe3','RBE3',[1 97 123456 1 123 98 1 123 99]);
fe_case(cf.mdl,'rbe3 -append','RBE3',[1 100 123456 1 123 101 1 123 102]);
fecom addRbe3
```

4.4.6 Viewing shapes

feplot displays shapes and color fields at nodes. The basic def data structure provides shapes in the .def field and associates each value with a .DOF (see mdof). For other inits see fecom InitDef.

```
[model,def]=demosdt('Demo gartfe'); % Get example
cf=feplot(model,def);  % display model and shapes
fecom('ch7');          % select channel 7 (first flex mode)
fecom('pro');          % Show model properties
```

Scan through the various deformations using the +/- buttons/keys or clicking 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 (see section 4.6 and fe_sens) using a stick or arrow sensor visualization (fecom showsens or fecom showarrow). On such plots, you can label some or all degrees of freedom using the call fecom ('doftext',idof).

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

Superposing shapes

Modeshape superposition is an important application (see plot of section 2.7.1) 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

```
[model,def]=demosdt('demo gartfe');cf=feplot(model); % demo init
cf.def(1)=def;  % First set of deformations
def.def=def.def+rand(size(def.def))/5;
cf.def(2)=def;  % second set of deformations
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

```
cf=demosdt('demo gartfe plot');
fecom(';showline; ch7 10')
```

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

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.

```
cf=demosdt('demo gartfe plot');
```

```
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 `FindNode` commands to display some node numbers. For example try `fecom('textnode egid<0 & y>0')`.

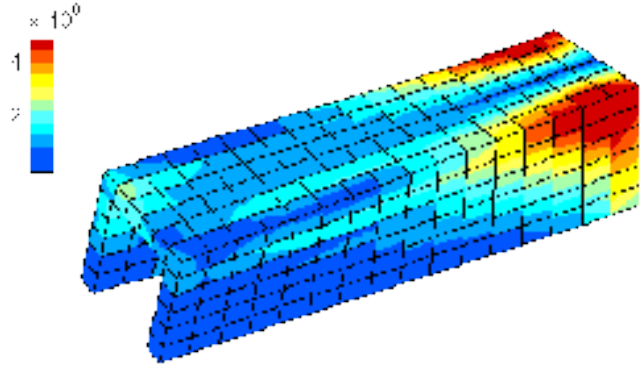


Figure 4.4: Stress level plot.

4.4.7 Viewing property colors

For reference information on colors, see `fecom ColorData`.

When preparing a model, one often needs to visualize property colors.

```
cf=feplot(demosdt('demogartfe'));
fecom('ColorDataMat'); % Display color associated with MatId
% Now a partial selection with nicer transparency
cf.sel={'eltname~=mass','ColorDataPro-alpha.1-edgealpha .05'}
```

How do I keep group colors constant when I select part of a model?

One can define different types of color for selection using `fecom ColorData`. In particular one can color by `GroupId`, by `ProId` or by `MatId` using respectively `fecom colordatagroup`, `colordatapro` or `colordatamat`. Without second argument, colors are attributed automatically. One can define a color map with each row of the form `[ID Red Green Blue]` as a second argument: `fecom('colordata',colormap)`. All `ID` do not need to be present in `colormap` matrix (colors for missing `ID` are then automatically attributed). Following example defines 3 color views of the same GART model:

```
cf=demosdt('demo gartFE plot');
```

```
% ID Red Green Blue
r1=[(1:10)' [ones(3,1); zeros(7,1)] ...
    [zeros(3,1); ones(7,1)] zeros(10,1)]; % colormap
fecom('colordatagroup',r1) % all ID associated with color
% redefine groups 4,5 color
cf.Stack{'GroupColor'}(4:5,2:4)=[0 0 1;0 0 1];
fecom('colordatagroup');
% just some ID associated with color
fecom('colordatapro',[1 1 0 0; 3 1 0 0])
fecom('colordatamat') % no color map defined
cf.Stack
```

4.4.8 Viewing colors at nodes

Color at nodes can be based on the current display. In particular, `ColorDataEvalA`, `EvalX`, ... `EvalRadZ`, `EvalTanZ` use the information of current motion from initial position to generate a color field dynamically. The advantage of this strategy is that no prior computation is needed.

Display of specific fields is another common application. Thus `ColorDataDOF 19` displays DOF .19 (pressure). This the field is not needed to display the motion of nodes, prior extraction from the deformations is needed.

4.4.9 Viewing colors at elements

Display of energies is a typical case of color at elements. Since computing energies for many deformations can take time, it is considered best practice to compute energies first and display energies next.

```
cf=demosdt('demo gartFE plot');
% If EltId are not consistent you may need to fix them
% The ; in 'eltidfix;' is used to prevent display of warning messages
[eltid,cf.mdl.Elt]=feutil('eltidfix;',cf.mdl);
Ek=fe_stress('Enerk -curve',cf.mdl,cf.def);
fecom(cf,'ColorDataElt',Ek)          % Values for each element
% Sum by group
fecom(cf,'ColorDataElt -bygroup -frac -colorbartitle "Frac %"',Ek)
```

More details are given in `fe_stress` `feplot`.

4.4.10 feplot FAQ

feplot lets you define and save advanced views of your model, and export them as .png pictures.

- **How do I display part of the model as wire frame? (Advanced object handling)**

  What is displayed in a feplot figure is defined by a set of objects. Once you have plotted your model with cf=feplot(model), you can access to displayed objects through cf.o(i) (i is the number of the object). Each object is defined by a selection of model elements ('seli') associated to some other properties (see fecom SetObject). Selections are defined as FindElt commands through cf.sel(i). Displayed objects or selections can be removed using cf.o(i)=[] or cf.sel(i)=[].
  Following example loads ubeam model, defines 2 complementary selections, and displays the second as a wire frame (ty2):

```
model=demosdt('demoubeam'); cf=feplot
% define visualisation
cf.sel(1)='WithoutNode{z>1 & z<1.5}';
cf.sel(2)='WithNode{z>1 & z<1.5}';
cf.o(1)={'sel1 ty1','FaceColor',[1 0 0]}; % red patch
cf.o(3)={'sel2 ty2','EdgeColor',[0 0 1]}; % blue wire frame
% reinit visualisation :
cf.sel(1)='groupall';
cf.sel(2)=[]; cf.o(3)=[];
```

- **Is feplot able to display very large models?**

  There is no theoretical size limitation for models to be displayed. However, due to the use of Matlab figures, and although optimization efforts have been done, feplot can be very slow for large models. This is due to the inefficient use of triangle strips by the Matlab calls to OpenGL, but to ensure robustness SDT still sticks to strict Matlab functionality for GUI operation.

  When encountering problems, you should first check that you have an appropriate graphics card, that has a large memory and supports OpenGL and that the Renderer is set to opengl. Note also that any X window forwarding (remote terminal) can result in very slow operation: large models should be viewed locally since Matlab does not support an optimized remote client.

  To increase fluidity it is possible to reduce the number of displayed patches using fecom command SelReduce*rp* where *rp* is the ratio of patches to be kept. Adjusting *rp*, fluidity can be significantly improved with minor visual quality loss.
  Following example draws a 50x50 patch, and uses fecom('ReduceSel') to keep only a patch out of 10:

```
model=feutil('ObjectQuad',[-1 -1 0;-1 1 0;1 1 0;1 -1 0],50,50);
cf=feplot(model); fecom(cf,'showpatch');
fecom(cf,'SelReduce .1'); % keep only 10% of patches.
```

If you encounter memory problems with `feplot` consider using `fecom load-hdf`.

- **How do I save figures?**

  You should not save `feplot` figures but models using `fecom Save`.

  To save images shown in `feplot`, you should see `iicom ImWrite`. If using the MATLAB print, you should use the `-noui` switch so that the GUI is not printed. Example `print -noui -depsc2 FileName.eps`.

- MATLAB gives the warning **Warning: RGB color data not yet supported in Painter's mode**. This is due to the use of true colors for `ColorDataMat` and other flat colors. You should save your figure as a bitmap or use the `fecom ShowLine` mode.

- **How do I define a colorbar scale and keep it constant during animation?**

  When using `fecom ColorDataEval` commands (useful when displayed deformation is restituted from reduced deformation at each step), color scaling is updated at each step.
  One can use `fecom('ScaleColorOne')` to force the colorbar scale to remain constant. In that case one can define the limit of the color map with `set(cf.ga,'clim',[-1 1])` where `cf` is a pointer to target feplot figure, and -1 1 can be replaced by color map boundaries.

- **How do I make an animation based on my deformation field displayed in feplot ?**

  Several strategies are available depending on the user needs.

  - The simplest way to do this is to generate an `avi` file using the `feplot` figure menu: `Feplot > Anim > MakeAVI`. Equivalent command line inputs with variants are provided in `fecom AnimMovie` documentation.
  - SDT allows generating animated `gif` from `feplot` animations using the `convert` function. `convert('AnimMovie25')` will generate a 25 steps `feplot` animation as an animated `gif`. To pilot a subsampling of steps, see `fecom Anim`. Note that the `convert` function is a gateway function to the `convert` function of `ImageMagick`, that should be installed on your system. You can look up `http://www.imagemagick.org` for more information.
  - Better `avi` results can be obtained in recent MATLAB by using the `VideoWriter` object with lower level `feplot` calls. The following code allows doing this

    ```
    writerObj = VideoWriter(['TEST2_ANIM.avi']); %'Archival');
     writerObj.FrameRate=830; % fps
     writerObj.Quality=100;
     open(writerObj);
    ```

```
cf.ua.PostFcn=sprintf(['evalin(''base'','...
'''frame = getframe(gcf);writeVideo(writerObj,frame);'')']);
frame = getframe;
writeVideo(writerObj,frame); % frame will contain the film
close(writerObj);
```

4.5 Other information needed to specify a problem

Once the mesh defined, to prepare analysis one still needs to define

- material and element properties associated to the various elements.
- boundary conditions, constraints (see section 4.5.4) and applied loads (see section 4.5.5)

Graphical editing of case properties is supported by the case tab of the model properties GUI (see section 4.5.3). The associated information is stored in a case data structure which is an entry of the `.Stack` field of the model data structure.

4.5.1 Material and element properties

You can edit material properties using the `Mat` 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, linear acoustic fluids, piezo-electric volumes, etc.

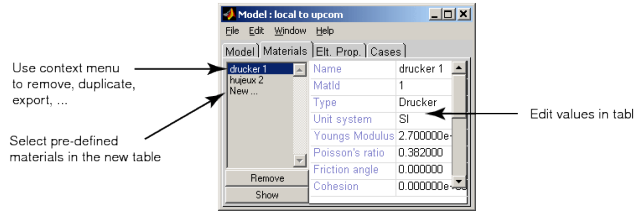


Figure 4.5: Material tab.

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

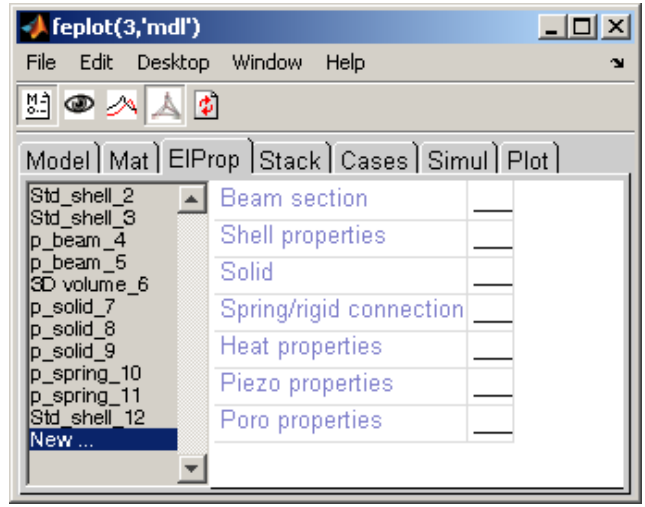


Figure 4.6: Property tab.

When the view mode is selected (icon pressed), you can see the elements affected by each material or element property by selecting it in the associated tab.

You can edit properties using the `Pro` tab of the `Model Properties` figure which lists current properties and lets you choose new ones from the database of each property type (Figure 4.6).

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

```
femesh('reset');
model=femesh('test 2bay plot');
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 ; ...
      p_beam('dbval 2','circle 4e-3') ; ... % circular section 4 mm
      p_beam('dbval 3','rectangle 4e-3 3e-3')...% rectangular section
 ];
```

Unit system conversion is supported in property definitions, through two command options.

- `-unit` command option asks for a specific unit system output. It thus expects possible input data in SI, prior to converting (and generating a proper `typ` value).

- `-punit` command option tells the function that a specific unit system is used. It thus expects possible input data in the specified unit system, and generates a proper `typ` value.

The 3 following calls are thus equivalent to define a beam of circular section of 4mm in the MM unit system:

```
il = p_beam('dbval -unit MM 2 circle 4e-3'); % given data in SI, output in MM
il = p_beam('dbval -punit MM 2 circle 4'); % given data in MM, output in MM
il = p_beam('dbval -punit CM -unit MM circle 0.4'); % given data in CM, output in 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 selections based on `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)`.

4.5.2 Other information stored in the stack

The stack can be used to store many other things (options for simulations, results, ...). More details are given in section 7.7 . You can get a list of current default entry builders with `fe_def('new')`.

```
info,    EigOpt, sdtdef('DefaultEigOpt-safe',[5 20 1e3])
info,      Freq, sdtdef('DefaultFreq-safe',[1:2])
 sel,       Sel, struct('data','groupall','ID',1)
...
```

4.5.3 Cases GUI

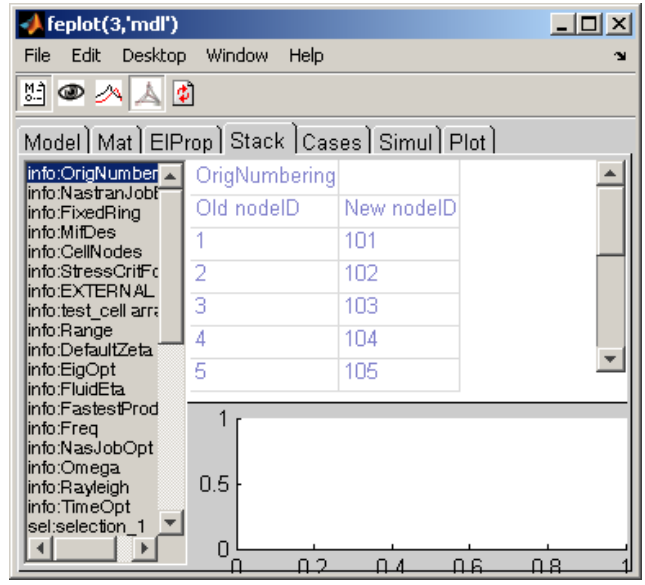


Figure 4.7: Stack tab.

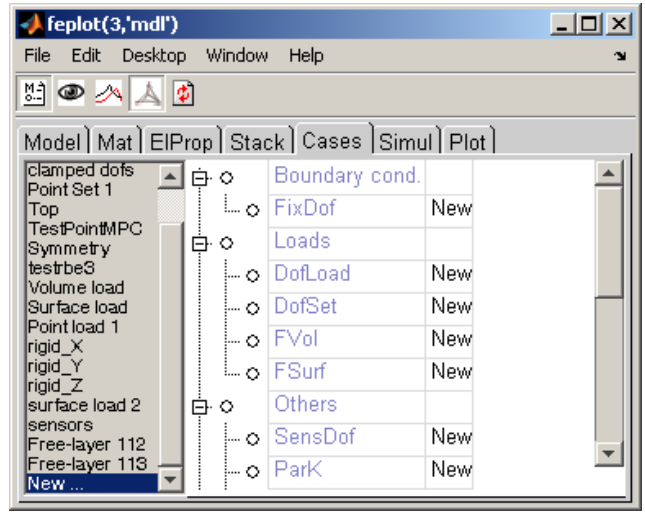


Figure 4.8: Cases properties tab.

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.

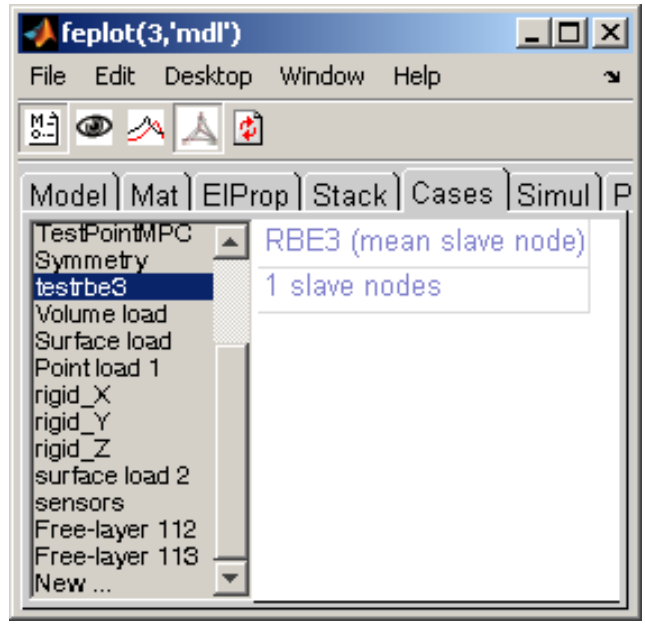


Figure 4.9: Cases properties tab.

4.5.4 Boundary conditions and constraints

Boundary conditions and constraints are described in in `Case.Stack` using `FixDof`, `Rigid`, ... case entries (see `fe_case` and section 7.7). (`KeepDof` still exists but often leads to misunderstanding)

`FixDof` entries are used to easily impose zero displacement on some DOFs. To treat the two bay truss example of section 4.1.1 , one will for example use

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

When assembling the model with the specified `Case` (see section 4.5.3), 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. FindNode commands allow node selection and fe_c provides additional DOF selection capabilities. Details on low level handling of fixed boundary conditions and constraints are given in section 7.14 .

4.5.5 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

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

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

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

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

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

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

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

4.6 Sensors

Sensors are used for test/analysis correlation and in analysis for models where one wants to post-process partial information by using an observation equation $\{y\} = [c]\{q\}$. This general objective is supported by the use of `SensDof` entries. This section addresses the following issues

- translation measurements associated simplified views (often wire-frame) is classical for modal testing and FEM post-processing. These can be simply defined using a `.tdof` field, see also section 2.7.1 and section 2.7.2 for wire frame geometry and sensor declaration. Commands `trans`, `triax` and `laser` provide simplified calls to generate the associated translation sensors.
- other sensor types typically used in analysis are
  - `rel` relative displacement sensor.
  - `general` general sensor (low level).
  - `resultant` resultant force sensor.
  - `strain` strain or stress sensor.
- topology correlation is the process in which sensor output is related to the DOFs of the underlying FEM. This is implemented as the `SensMatch` command detailed section 4.6.4 . In the case of translation measurements, this is only needed for test/analysis correlation.

4.6.1 Sensor GUI, a simple example

Using the feplot properties GUI, one can edit and visualize sensors. The following example loads `ubeam` model, defines some sensors and opens the sensor GUI.

```
model=demosdt('demo ubeam-pro');
cf=feplot; model=cf.mdl;

model=fe_case(model,'SensDof append trans','output',...
    [1,0.0,0.5,2.5,0.0,0.0,1.0]); %  add a translation sensor
```

```
model=fe_case(model,'SensDof append triax','output',8); % add triax sensor
model=fe_case(model,'SensDof append strain','output',...
    [4,0.0,0.5,2.5,0.0,0.0,1.0]); %  add strain sensor

model=fe_case(model,'sensmatch radius1','output'); % match sensor set 'output'

fecom(cf,'promodelviewon');
fecom(cf,'curtab Cases','output'); % open sensor GUI
```

Clicking on `Edit Label` one can edit the full list of sensor labels.
The whole sensor set can be visualized as arrows in the feplot figure clicking on the eye button on the top of the figure. Once visualization is activated one can activate the cursor on sensors by clicking on `CursorSel`. Then one can edit sensor properties by clicking on corresponding arrow in the feplot figure.

The icons in the GUI can be used to control the display of wire-frame, arrows and links.

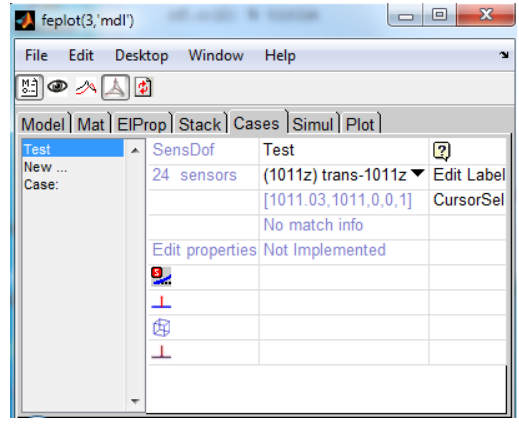


Figure 4.10: GUI for sensor edition

4.6.2 Sensor definition from a cell array

Experimental setups can be defined with a cell array containing all the information relative to the sensors (only displacement/velocity/acceleration sensors are currently supported). This array is meant to be filled any table editor, possibly outside MATLAB. Using EXCEL you can read it with `data=sdtacx('excel read filename',sheetnumber)`.

The first row gives column labels (the order in which they are given is free). Each of the following

rows defines a sensor. Known column headers are

- `'lab'` contains the names of the sensors. Providing a name for each sensor is mandatory.
- `'SensType'` contains optional information such as the name of the sensor manufacturer, their types, *etc.*
- `'SensId'` contains the identification numbers of the sensors. Each sensor must have a **unique** `SensId`. If the identification is non integer, the integer part is taken to be a `NodeId`. For example `10.01` will be taken to be node 10.
- `'X'`, `'Y'` and `'Z'` contain the cartesian coordinates of each sensor in the **reference frame**. For cylindrical coordinates replace the column headers by `'R'`, `'Theta'` and `'Z'` (mixing both types of coordinates inside the cell array is not currently supported). Such columns are mandatory except of localization is given by `FEMId`.
- `'FEMId'` can be used to specify localization and help node matching.
- `'DirSpec'` contains a specification of the direction in which the measurement is done at each sensor. A minus in front of any specification can be used to generate the opposite direction (`-TX` for example). Available entries are

'dir *x y* '	Direction of measurement specified trough its components in global coordinates (the vector is normalized).
'X'	[1 0 0], in the reference frame
'Y'	[0 1 0], in the reference frame
'Z'	[0 0 1], in the reference frame
'N'	normal to the element(s) to which the sensor is matched (automatically detected in the subsequent call to SensMatch)
'TX'	tangent to matched surface in the N, X plane.
'TY'	tangent to matched surface in the N, Y plane
'TZ'	tangent to matched surface in the N, Z plane
'N^TX'	tangent orthogonal to the N, X plane
'N^TY'	tangent orthogonal to the N, Y plane
'N^TZ'	tangent orthogonal to the N, Z plane
'laser xs ys zs'	where (x_s, y_s, z_s) are the coordinates of the primary or secondary source (when mirrors are used).
'FEM 10.01'	associated FEM DOF

triax sensors are dealt with by defining three sensors with the same 'lab' but different 'SensId' and 'DirSpec'. In this case, a straightforward way to define the measurement directions is to make the first axis be the normal to the matching surface. The second axis is then forced to be parallel to the surface and oriented along a preferred reference axis, allowed by the possibility to define 'T*'. The third axis is therefore automatically built so that the three axes form a direct orthonormal basis with a specification such as N^T*. Note that there is no need to always consider the orthonormal basis as a whole and a single trans sensor with either 'T*' or N^T* as its direction of measure can be specified.

In the example below, one considers a pentahedron element and aims to observe the displacement just above the slanted face. The first vector is the normal to that face whose coordinates are

$[-\sqrt{2}/2, \sqrt{2}/2, 0]$. The second one is chosen (i.) parallel to the observed face, (ii.) in the (x, y) plane and (iii.) along x axis, so that its coordinates are $[\sqrt{2}/2, \sqrt{2}/2, 0]$. Finally, the coordinates of the last vector can only be $[0, 0, -1]$ to comply with the orthonormality conditions. The resulting sensor placement is depicted in figure 4.11

```
cf=feplot;cf.model=femesh('testpenta6');fecom('triax');

% sensor definition as cell array
tcell={'lab', 'SensType','SensId','X','Y','Z','DirSpec';...
      'sensor 1','',      1.02,.4,.6,.5,'N';
      'sensor 2','',      1.01,.4,.6,.5,'TX';
      'sensor 3','',      2.01,.4,.6,1.,'dir 1 -1 1';
      'sensor 4','',      1.09,.4,.6,.5,'N^TX'
      'sensor 5','',      3.01,[],[],[],'FEM 5.01'
      'sensor 6','',      4.02, 1, 0, 1,'Y'
      };disp(tcell)
%sens=fe_sens('tdoftable',tcell);
cf.mdl=fe_case(cf.mdl,'SensDof','Test',tcell);
cf.mdl=fe_case(cf.mdl,'SensMatch radius1','Test','selface');
fecom(cf,'curtab Cases','Test'); fecom(cf,'ProViewOn')% open sensor GUI
sens=fe_case(cf.mdl,'sens');
fe_sens('tdoftable',cf,'Test') % see summary of match results
fname=fullfile(sdtdef('tempdir'),'SensSpec.xls');
if ~isunix % Test write to excel to illustrate ability to reread
 xlswrite(fname,tcell,'Sensors');
 sdtweb('_link',sprintf('open(''%s'')',fname))
end
```

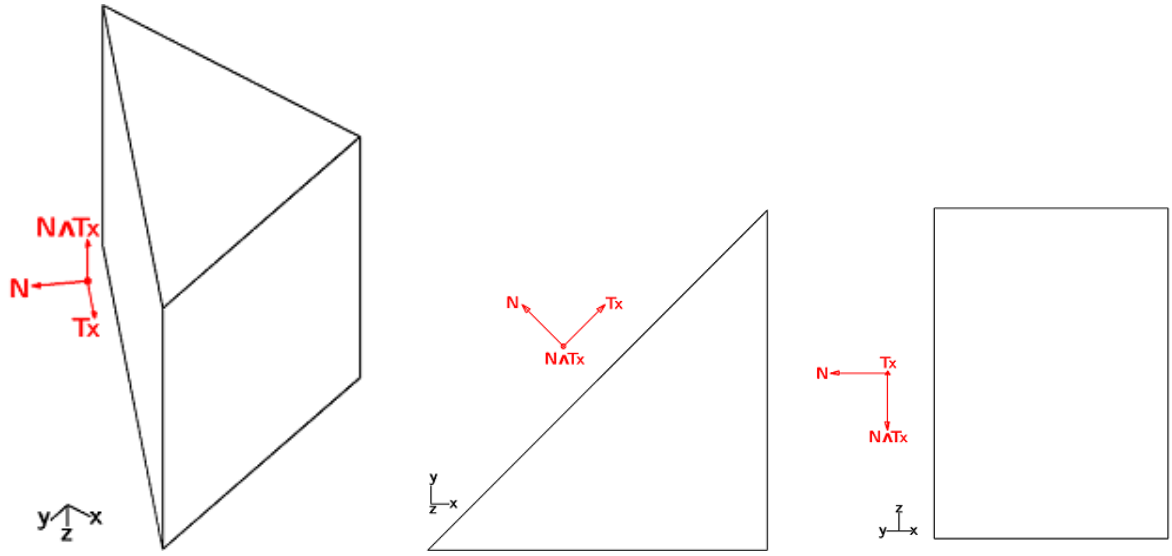


Figure 4.11: Typical axis definition of a triax sensor attached to a penta6

It is now possible to generate the experimental setup of the ubeam example described in the previous section by the means of a single cell array containing the data relative to both the trans and triax sensors.

```
model=demosdt('demo ubeam-pro');
cf=feplot; model=cf.mdl;
n8=feutil('getnode NodeId 8',model); % triax pos.
tdof={'lab','SensType','SensId','X','Y','Z','DirSpec';...
      'sensor1 - trans','',1,0.0,0.5,2.5,'Z';
      'sensor2 - triax','',2,n8(:,5),n8(:,6),n8(:,7),'X';
      'sensor2 - triax','',3,n8(:,5),n8(:,6),n8(:,7),'Y';
      'sensor2 - triax','',4,n8(:,5),n8(:,6),n8(:,7),'Z'};
sens=fe_sens('tdoftable',tdof);
cf.mdl=fe_case(cf.mdl,'SensDof','output',sens);
cf.mdl=fe_case(cf.mdl,'SensMatch radius1');
fecom(cf,'curtab Cases','output'); % open sensor GUI
```

4.6.3 Sensor data structure and init commands

This is a reference section on SensDof case entries. A tutorial on the basic configuration with a test wire frame and translation sensors is given in section 2.7 . SensDof entries can contain the following

fields

`sens.Node`	(optional) node matrix for sensor nodes that are not in the model. When defined, all node numbers in `sens.tdof` should refer to these nodes. The order typically differs from that in `.tdof`, you can get the positions with `fe_sens('tdofNode',model,SensName)`.
`sens.Elt`	element description matrix for a wire-frame display of the sensors (typically for test wire-frames).
`sens.bas`	Coordinate system definitions for `sens.Node`, see `fe_sens basis`
`sens.tdof`	see details below.
`sens.DOF`	DOF definition vector for the analysis (finite element model). It defines the meaning of columns in `sens.cta`.
`sens.cta`	is an observation matrix associated with the observation equation $\{y\} = [c]\{q\}$ (where q is defined on `sens.DOF`). This is built using the `fe_case sens` command illustrated below.
`sens.Stack`	cell array with one row per sensor giving `'sens','SensorTag',data` with `data` is a structure. `SensorTag` is obtained from `SensId` (first column of `tdof`) using `feutil('stringdof',SensId)`. It is used to define the tag uniquely and may differ from the label that the user may want to associated with a sensor which is stored in `data.lab`.

The `sens.tdof` field declares translation sensors and their directions

- nominally is 5 column matrix with rows containing `[SensID NodeID nx ny nz]` giving a sensor identifier (integer or real), a node identifier (positive integer, if relevant), a direction.
- can be single column DOF definition vector which can be transformed to 5 column format using `tdof = fe_sens('tdof',sens.tdof)`
- `SensId` gives an identifier for each sensor. It should thus be unique and there may be conflicts if it is not.
- `NodeId` specifies a node identifier for the spatial localization of the sensor. If not needed (resultant sensors for example), `NodeId` can be set for zero.

  `NodeId>0` corresponds is for use of `model.Node` locations and sens.Node should not be defined.

  `NodeId<0` is used to look for the node position in `sens.Node` rather than `model.Node`. Mixed definitions (some NodeId positive and other negative) are not supported.

  Most initialization calls accept the specification of a physical `x y z` position, a `.vert0` field is then defined.
- `nx ny nz` specifies a measurement direction for sensors that need one.

All sensors are generated with the command
`fe_case(model,'SensDof <append, combine> Sensor_type',Sensor,data,SensLab)`
`Sensor` is the case entry name to which sensors will be added. `data` is a structure, a vector, or a matrix, which describes the sensor to be added. The nature of `data` depends on `Sensor_type` as detailed below. `SensLab` is an optional cell array used to define sensor labels. There should be as much elements in `SensLab` as sensors added. If there is only one string in the cell array `SensLab`, it is used to generate labels substituting for each sensor `$id` by its SensId, `$type` by its type (trans, strain ...), `$j1` by its number in the set currently added. If `SensLab` is not given, default label generation is `$type_$id`.
In the default mode (`'SensDof'` command), new sensors replace any existing ones. In the append mode (`'SensDof append'`), if a sensor is added with an existing `SensID`, the `SensID` of new sensor will changed to a free `SensID` value. In the combine mode (`'SensDof combine'`), existing sensor with the same `SensID` will be replaced by the new one.

rel

Relative displacement sensor or relative force sensor (spring load). Data passed to the command is `[NodeID1 NodeID2]`.

This sensor measures the relative displacement between `NodeID1` and `NodeID2`, along the direction defined from `NodeID1` to `NodeID2`. One can use the command option `-dof` in order to measure along the defined DOF directions (mandatory if the two nodes are coincident). As many sensors as DOF are then added. For a relative force sensor, on can use the command option `-coef` to define the associated spring stiffness (sensor value is the product of the relative displacement and the stiffness of the spring).

If some DOF are missing, the sensor will be generated with a warning and a partial observation corresponding to the found DOF only.

The following example defines 3 relative displacement sensors (one in the direction of the two nodes, and two others along x and y):

```
model=demosdt('demo ubeam-pro')
data=[30 372];
model=fe_case(model,'SensDof append rel','output',data);
model=fe_case(model,'SensDof append rel -dof 1 2','output',data);
```

general

General sensors are defined by a linear observation equation. This is a low level definition that should be used for sensors that can't be described otherwise. Data passed to the command is a

structure with field `.cta` (observation matrix), `.DOF` DOF associated to the observation matrix, and possibly `.lab` giving a label for each row of the observation matrix.

The following example defines a general sensor

```
model=demosdt('demo ubeam-pro');
Sensor=struct('cta',[1 -1;0 1],'DOF',[8.03; 9.03]);
model=fe_case(model,'SensDof append general','output',Sensor);
```

trans

Translation sensors (see also section 2.7.2) can be specified by giving

```
[DOF]
[DOF, BasID]
[SensID, NodeID, nx, ny, nz]
[SensID, x, y, z, nx, ny, nz]
```

This is often used with wire frames, see section 2.7.2 . The definition of **test** sensors is given in section 3.1.1 .

The basic case is the measurement of a translation corresponding the main directions of a coordinate system. The `DOF` format (1.02 for $1y$, see section 7.5) can then be simply used, the DOF values are used as is then used as `SensID`. Note that this form is also acceptable to define sensors for other DOFs (rotation, temperature, ...).

A number of software packages use local coordinate systems rather than a direction to define sensors. SDT provides compatibility as follows.

If `model.bas` contains local coordinate systems and deformations are given in the global frame (`DID` in column 3 of `model.Node` is zero), the directions `nx ny nz` (`sens.tdof` columns 3 to 5) must reflect local definitions. A call giving `[DOF, BasID]` defines the sensor direction in the main directions of basis `BasID` and the sensor direction is adjusted.

If FEM results are given in local coordinates, you should not specify a basis for the sensor definition, the directions `nx ny nz` (`sens.tdof` columns 3 to 5) should be `[1 0 0]`, ... as obtained with a simple `[DOF]` argument in the sensor definition call.

When specifying a `BasId`, it the sensor direction `nx ny nz` is adjusted and given in global FEM coordinates. Observation should thus be made using FEM deformations in global coordinates (with a `DID` set to zero). If your FEM results are given in local coordinates, you should not specify a basis for the sensor definition. You can also perform the local to global transformation with

```
cGL= basis('trans E',model.bas,model.node,def.DOF)
def.def=cGL*def.def
```

The last two input forms specify location as `x y z` or `NodeID`, and direction `nx ny nz` (this vector need not be normalized, sensor value is the scalar product of the direction vector and the displacement vector).

One can add multiple sensors in a single call `fe_case(model,'SensDof <append> trans', Name, Sensor)` when rows of sensors contain sensor entries of the same form.

Following example defines a translation sensor using each of the forms

```
model=demosdt('demo ubeam-pro')
model.bas=basis('rotate',[],'r=30;n=[0 1 1]',100);
model=fe_case(model,'SensDof append trans','output',...
  [1,0.0,0.5,2.5,0.0,0.0,1.0]);
model=fe_case(model,'SensDof append trans','output',...
  [2,8,-1.0,0.0,0.0]);
model=fe_case(model,'SensDof append trans','output',...
  [314.03]);
model=fe_case(model,'SensDof append trans','output',...
  [324.03 100]);
cf=feplot;cf.sel(2)='-output';cf.o(1)={'sel2 ty 7','linewidth',2};
```

`Sens.Stack` entries for translation can use the following fields

`.vert0`	physical position in global coordinates.
`.ID`	`NodeId` for physical position. Positive if a model node, negative if `SensDof` entry node.
`.match`	cell array describing how the corresponding sensor is matched to the reference model. Columns are `ElemF,elt,rstj,StickNode`.

dof

One can simply define a set of sensors along model DOFs with a direct `SensDof` call `model=fe_case(model,'SensDof','SensDofName',DofList)`. There is no need in that case to pass through `SensMatch` step in order to get observation matrix.

```
model=demosdt('demo ubeam-pro')
model=fe_case(model,'SensDof','output',[1.01;2.03;10.01]);
Sens=fe_case(model,'sens','output')
```

triax, laser

A triax is the same as defining 3 translation sensors, in each of the 3 translation DOF (0.01, 0.02 and 0.03) of a node. Use `fe_case(model,'SensDof append triax', Name, NodeId)` with a vector `NodeId` to add multiple triaxes. A positive `NodeId` refers to a FEM node, while a negative refers to a wire frame node.

For scanning laser vibrometer tests
`fe_sens('laser px py pz',model,SightNodes,'SensDofName')`
appends translation sensors 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'` or also giving a vector of NodeId. If a test wire frame exists in the `SensDofName` entry, node selection command or NodeId list are defined in this model. If you want to flip the measurement direction, use a call of the form

```
cf.CStack{'output'}.tdof(:,3:5)=-cf.CStack{'output'}.tdof(:,3:5)
```

The following example defines some laser sensors, using a test wire frame:

```
cf=demosdt('demo gartfeplot'); model=cf.mdl;% load FEM
TEST=demosdt('demo garttewire');  % see sdtweb('pre#presen')
TEST.tdof=[];%Define test wire frame, but start with no tdof
model=fe_case(model,'SensDof','test',TEST)
model=fe_case(model,'SensDof Append Triax','test',-TEST.Node(1))

% Add sensors on TEST wire frame location
model=fe_sens('laser 0 0 6',model,-TEST.Node(2:end,1),'test');
% Show result
fecom('curtab Cases','output'); fecom('proviewon');
```

To add a sensor on FEM node you would use `model=fe_sens('laser 0 0 6',model,20,'test');` but this is not possible here because `SensDof` entries do not support mixed definitions on test and FEM nodes.

strain,stress

Note that an extended version of this functionality is now discussed in section 4.7 . Strain sensors can be specified by giving

```
[SensID, NodeID]
[SensID, x, y, z]
[SensID, NodeID, n1x, n1y, n1z]
```

```
[SensID, x, y, z, n1x, n1y, n1z]
[SensID, NodeID, n1x, n1y, n1z, n2x, n2y, n2z]
[SensID, x, y, z, n1x, n1y, n1z, n2x, n2y, n2z]
```

when no direction is specified 6 sensors are added for stress/strains in the x, y, z, yz, zx, and xy directions (`SensId` is incremented by steps of 1). With `n1x n1y n1z` (this vector need not be normalized) on measures the axial strain in this direction. For shear, one specifies a second direction `n2x n2y n2z` (this vector need not be normalized) (if not given n_2 is taken equal to n_1). The sensor value is given by $\{n_2\}^T [\epsilon] \{n_1\}$.
`Sensor` can also be a matrix if all rows are of the same type. Then, one can add a set of sensors with a single call to the `fe_case(model,'SensDof <append> strain', Name, Sensor)` command.

Following example defines a strain sensor with each possible way:

```
model=demosdt('demo ubeam-pro')
model=fe_case(model,'SensDof append strain','output',...
  [4,0.0,0.5,2.5,0.0,0.0,1.0]);
model=fe_case(model,'SensDof append strain','output',...
  [6,134,0.5,0.5,0.5]);
model=fe_case(model,'SensDof append strain','output',...
  [5,0.0,0.4,1.25,1.0,0.0,0.0,0.0,0.0,1.0]);
model=fe_case(model,'SensDof append strain','output',...
  [7,370,0.0,0.0,1.0,0.0,1.0,0.0]);
```

Stress sensor.
It is the same as the strain sensor. The sensor value is given by $\{n_2\}^T [\sigma] \{n_1\}$.
Following example defines a stress sensor with each possible way:

```
model=demosdt('demo ubeam-pro')
model=fe_case(model,'SensDof append stress','output',...
  [4,0.0,0.5,2.5,0.0,0.0,1.0]);
model=fe_case(model,'SensDof append stress','output',...
  [6,134,0.5,0.5,0.5]);
model=fe_case(model,'SensDof append stress','output',...
  [5,0.0,0.4,1.25,1.0,0.0,0.0,0.0,0.0,1.0]);
model=fe_case(model,'SensDof append stress','output',...
  [7,370,0.0,0.0,1.0,0.0,1.0,0.0]);
```

Element formulations (see section 6.1) include definitions of fields and their derivatives that are strain/stress in mechanical applications and similar quantities otherwise. The general formula is $\{\epsilon\} = [B(r,s,t)] \{q\}$. These (generalized) strain vectors are defined for all points of a volume and the default is to use an exact evaluation at the location of the sensor.

In practice, the generalized strains are more accurately predicted at integration points. Placing the sensor arbitrarily can generate some inaccuracy (for example stress and strains are discontinuous across element boundaries two nearby sensors might give different results). The `-stick` option can be used to for placement at specific gauss points. `-stick` by itself forces placement of the sensor and the center of the matching element. This will typically be a more appropriate location to evaluate stresses or strains.

To allow arbitrary positioning some level of reinterpolation is needed. The procedure is then to evaluate strain/stresses at Gauss points and use shape functions for reinterpolation. The process must however involve multiple elements to limit interelement discontinuities. This procedure is currently implemented through the `fe_caseg('StressCut')` command, as detailed in section 4.7 .

`resultant`

Resultant sensors measure the resultant force on a given surface. **Note** that the observation of resultant fields is discussed in section 4.7.3 . They can be specified by giving a structure with fields

`.ID`	sensor ID.
`.EltSel`	`FindElt` command that gives the elements concerned by the resultant.
`.SurfSel`	`FindNode` command that gives the surface where the resultant is computed.
`.dir`	with 3 components direction of resultant measurement, with 6 origin and direction of resulting moment in global coordinates. This vector need not be normalized (scalar product). For non-mechanical DOF, `.dir` can be a scalar DOF (`.21` for electric field for example)
`.type`	contains the string 'resultant'.

Following example defines a resultant sensor:

```
model=demosdt('demo ubeam-pro')
Sensor.ID=1;
Sensor.EltSel='WithNode{z==1.25} & WithNode{z>1.25}';
Sensor.SurfSel='z==1.25';
Sensor.dir=[0.0 0.0 1.0];
Sensor.type='resultant';
model=fe_case(model,'SensDof append resultant','output',Sensor);
```

Resultant sensors are not yet available for superelements model.

4.6.4 Topology correlation and observation matrix

Sens, observation

This command is used after SensMatch to build the observation equation that relates the response at sensors to the response a DOFs

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

where the c matrix in stored in the sens.cta field and DOFs expected for q are given in sens.tdof.

After the matching phase, one can build the observation matrix with SensFull=fe_case(model,'sens',SensDofEntryName) or when using a reduced superelement model SensRed=fe_case(model,'sensSE',SensDofEntryName). Note that with superelements, you can also define a field .UseSE=1 in the sensor entry to force use of the reduced model. This is needed for the generation of reduced selections in feplot (typically cf.sel='-Test').

The following example illustrates nominal strategies to generate the observed shape, here for a static response.

```
model=demosdt('demoUbeamSens'); def=fe_simul('static',model);

% Manual observation, using {y} = [c] {q}
sens=fe_case(model,'sens');
def=feutilb('placeindof',sens.DOF,def); % If DOF numbering differs
% could use  sens=feutilb('placeindof',def.DOF,sens); if all DOF present
y=sens.cta*def.def
% Automated curve generation
C1=fe_case('sensObserve',model,'sensor 1',def)
```

SensMatch

Once sensors defined (see trans, ...), sensors must be matched to elements of the mesh. This is done using
`model = fe_case(model,'sensmatch',SensDofEntryName);`

You may omit to provide the name if there is only one sensor set. The command builds the observation matrix associated to each sensor of the entry Name, and stores it as a .cta field, and associated .DOF, in the sensor stack.

Storing information in the stack allows multiple partial matches before generating the global observation matrix. The observation matrix is then obtained using

```
Sens = fe_case(model,'sens',SensDofEntryName);
```

The matching operation requires finding the elements that contain each sensor and the position within the reference element shape so that shape functions can be used to interpolate the response. Typical variants are

- a `radius` can be specified to modify the default sphere in which a match is sought. This is typically needed in cases some large elements.

  ```
  model=fe_case(model,'sensmatch radius1.0',Name)
  ```

- elements on which to match can be specified as a `FindElt` string. In particular, matching nodes outside volumes is not accepted. To obtain a match in cases where test nodes are located outside volume elements, you must thus match on the volume surface using
  `fe_case(model,'sensmatch radius1.0',Name,'selface')`
  which selects external surface of volumes and allows a normal projection towards the surface and thus proper match of sensors outside the model volume.

  Note that this selection does not yet let you selected implicit elements within a superelement.

- Matching on elements is not always acceptable, one can then force matching to the closest node. `SensMatch-Near` uses the motion at the matched node. `SensMatch-Rigid` uses a rigid body constraints to account for the distance between the matched node and the sensor (but is thus only applicable to cases with rotations defined at the nearby node).

In an automated match, the sensor is not always matched to the correct 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 a selection to define element subset on which perform the match. If you want to match one or more specific sensors to specific element subset, you can give cell array with SensId of sensor to match in a first column and with element string selector in a second column.

```
model=fe_case(model,'SensMatch',Name,{SensIdVector,'FindEltString'});
```

This is illustrated below in forcing the interpolation of test node 1206 to use FEM nodes in the plane where it is glued.

```
cf=demosdt('demo gartte cor plot');
fe_case(cf,'sensmatch -near')
fecom('curtabCases','sensors');fecom('promodelviewon');
% use fecom CursorSelOn to see how each sensor is matched.
cf.CStack{'sensors'}.Stack{18,3}
% modify link to 1206 to be on proper surface
```

```
cf.mdl=fe_case(cf.mdl,'SensMatch-near',...
    'sensors',{1206.02,'withnode {z>.16}'});
cf.CStack{'sensors'}.Stack{18,3}
% force link to given node (may need to adjust distance)
cf.mdl=fe_case(cf.mdl,'SensMatch-rigid radius .5','sensors',{1205.07,21});
cf.CStack{'sensors'}.Stack{19,3}

fecom('showlinks sensors');fecom('textnode',[1206 1205])
```

DofLoadSensDof

The generation of loads is less general than that of sensors. As a result it may be convenient to use reciprocity to define a load by generating the collocated sensor. When a sensor is defined, and the topology correlation performed with SensMatch, one can define an actuator from this sensor using `model=fe_case(model,'DofLoad SensDof',Input_Name,'Sens_Name:Sens_Nb')` or for model using superelements
`model=fe_case(model,'DofLoad SensDofSE',Input_Name,'Sens_Name:Sens_Nb')`.
Sens_Name is the name of the sensor set entry in the model stack of the translation sensor that defines the actuator, and Sens_Nb is its number in this stack entry. Thus Sensors:1 2 5 will define actuators with sensors 1, 2 and 5 for SensDof entry Sensors. Input_Name is the name of the DofLoad entry that will be created in the model stack to describe the actuator.

Note that a verification of directions can be performed a posteriori using feutilb GeomRB.

Animation of sensor wire-frame models

This is discussed in section 2.7.3 .

Obsolete

SDT 5.3 match strategies are still available. Only the arigid match has not been ported to SDT 6.1. This section thus documents SDT 5.3 match calls.

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).

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

```
[model,def]=demosdt('demo gartte cor');
model=fe_sens('rigid sensors',model); % link sensors to model
% display sensor wire-frame and animate FEM modes
cf=feplot; cf.model=model; cf.sel='-sensors';
cf.def=def;fecom(';undefline;scd.5;ch7')
```

4.7 Stress observation

Observation of stress and resultant fields is an application that requires specific tools for performance. A number of commands are thus available for this purpose. The two main commands are `fe_caseg StressCut` for generation of the observation and `fe_caseg StressObserve` for the generation of a `curve Multi-dim curve` showing observations as a table.

This functionality has been significantly stabilized for SDT 6.5 but improvements and minor format changes are still likely for future releases.

4.7.1 Building view mesh

Stresses can be observed at nodes of arbitrary meshes (view meshes that are very much related to test wireframes). You should look-up `feutil('object')` commands for ways to build simple shapes. A few alternate model generation calls are provided in `fe_caseg StressCut` as illustrated below and in the example for resultant sensors.

```
% Build straight line by weighting of two nodes
VIEW=fe_caseg('stresscut', ...
  struct('Origin',[0 0 0;0 0 1], ... % [n1,n2]
  'steps',linspace(0,1,10)))

% Automated build of a cut (works on convex cuts)
model=demosdt('demoubeam-pro');cf=feplot;
RO=struct('Origin',[0 0 .5],'axis',[0 0 1]);
VIEW=fe_caseg('StressCut',RO,cf);
feplot(VIEW) % note problem due to non convex cut

%View at Gauss points
model=demosdt('demoubeam-pro');cf=feplot;
cut=fe_caseg('StressCut-SelOut',struct('type','Gauss'),model);
cuts=  fe_caseg('stresscutToStrain',cut);

% Observe beam strains at Gauss points
[model,def]=beam1t('testeig')
mo1=fe_caseg('StressCut',struct('type','BeamGauss'),model);
cut=fe_caseg('StressCut -radius 10 -SelOut',mo1,model);
C1=fe_caseg('StressObserve -crit""',cut,def) % Observation as CURVE
```

Generic command is :
`VIEW=fe_caseg('StressCut',RO,model);`
`RO` is a data structure defining the view mesh. Different views are available according to `RO.type` or `RO` fields:

- `RO.type='conform'` When one wants to define a mesh that is a subpart of the model, there is no need to perform the match step, and the type `'conform'` can be used. The selection of the subpart of the model is performed through a FindElt command provided in `RO.sel`.
- `RO.type='gauss'` gauss points of the elements. A FindElt command can be provided in `RO.sel` (if omitted, all Gauss point are computed). For mechanical problems, to obtain the displacement gradient rather than the usual strain set `il(6)=100`.
- `RO.type='beamgauss'` : gauss points of a beam model.
- Plane cut mesh. `RO.Origin` and `RO.axis` must be filled. Cut is done in the plane defined by `RO.Origin` and `RO.axis`. If `RO.planes` is defined, as many planes (orthogonal to axis) as positions from the `RO.Origin` are defined.

- Cut line : `RO.Origin` defining line extremities (each row defines an extremity position, 3 columns for X Y and Z) and `RO.steps` defining the number of observation nodes must be filled.

4.7.2 Building and using a selection for stress observation

The first use of `StressCut` is to build a `feplot` selection to be used to view/animate stress fields on the view mesh. A basic example is shown below.

```
% build model
model=demosdt('volbeam');cf=feplot(model);

% build view mesh
VIEW=fe_caseg('stresscut', ...
  struct('Origin',[0 .05 .05;1 .05 .05], ... % [n1,n2]
  'steps',linspace(1,0,10)))
% build stress cut view selection
sel=fe_caseg('stresscut -selout',VIEW,cf);cla(cf.ga);feplot % generation observation

cf.def=fe_eig(model,[5 10 0]);
fe_caseg('stresscut',sel,cf) % Overlay view and nominal mesh
fecom('scc2') % Force equal scaling
```

The result of `StressCut` is found in `sel.StressObs.cta` which is an observation matrix giving the linear relation between motion at DOF of the elements connected to target points, to stress components at these target points. The procedure used to build this observation matrix in `fe_caseg` is as follows

- match desired nodes to the interior of elements and keep the resulting element coordinates. One then adds to the selected element set, one layer of elements with the same material and property ID (all elements that have one node in common with the matched elements);
- generate stress observation at Gauss points of the selected elements;
- for each stress component compute the stress at nodes that would lead to the same values at Gauss points. In other words one resolves

$$\sum_g (w_g J_g \{N_i(g)\}^T \{N_j(g)\} \sigma_j) = \sum_g (w_g J_g \{N_i(g)\}^T \sigma_g) \tag{4.2}$$

- finally use the element shape functions to interpolate each stress component from nodal values to values at the desired points using element coordinates found at the first step.

Note that typically, a `sel.StressObs.trans` field gives the observation matrix associated with translations at the target points to allow animation of positions as well as colors.

4.7.3 Observing resultant fields

`StressCut` sensors provide stress post-treatments in model cutoffs. The command interprets a data structure with fields

`.EltSel`	`FindElt` command that gives the elements concerned by the resultant.
`.SurfSel`	`FindNode` command that gives the selection where the resultant is computed.
`.type`	contains the string `'resultant'`.

Following example defines a `StressCut` call to show modal stresses in an internal surface of a volumic model

```
demosdt('demoubeam')
cf=feplot;fecom('showpatch')
cf.mdl=feutil('lin2quad',cf.mdl); % better stress interpolation
def=fe_eig(cf.mdl,[5 10 1e3]);
cf.def=def;
r1=struct('EltSel','withnode {z<2}', ...
    'SurfSel','inelt{innode{z==2}}', ...
    'type','Resultant');
fe_caseg('stresscut',r1,cf);
% adapt transparencies
fecom(cf,'SetProp sel(1).fsProp','FaceAlpha',0.01,'EdgeAlpha',0.2);
```

The observation in `feplot` is performed on the fly, with data stored in `cf.sel(2).StressObs` (for the latter example).

Command option `-SelOut` allows recovering the observation data. Field `.cta` is here compatible with general sensors, for customized observation.

```
cta=fe_caseg('StressCut-SelOut',r1,cf);
```

4.8 Computing/post-processing the response

4.8.1 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.

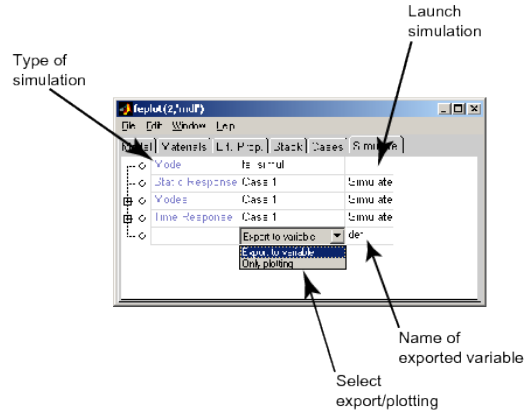


Figure 4.12: Simulation properties tab.

4.8.2 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');% 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);
```

4.8.3 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 4.3.2). 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).

4.8.4 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 4.8.3 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`).

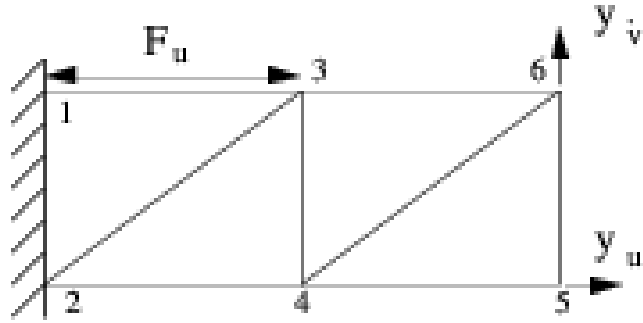


Figure 4.13: Truss example.

For the two bay truss examples 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)';
nor2xf(DEF,.01,Case,w,'hz iiplot "Main" -reset');
```

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');
```

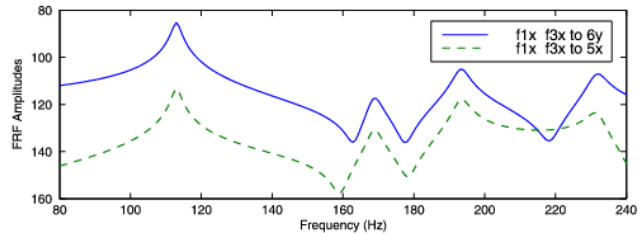


Figure 4.14: FRF synthesis : with and without static correction.

As detailed in section 6.2.3 , 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('demo ubeam sens -pro');

model=stack_set(model,'info','Freq',linspace(10,1e3,500)');
model=stack_set(model,'info','DefaultZeta',.01);

[SYS,T]=fe2ss('free 6 10',model); %ii_pof(eig(SYS.a),3)

qbode(SYS,linspace(10,1e3,1500)'*2*pi,'iiplot "Initial" -reset');
nor2xf(T,[.04],model,'hz iiplot "Damped" -po');
```

computes 10 modes using a full solution (*Eigopt*=[6 10 0]), appends the static response to the defined loads, and builds the state-space model corresponding to modal truncation with static correction (see section 6.2.3). **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`.

4.8.5 Viewing shapes, stress, energy, ...

NEED TO INTRODUCE PROPER REFERENCES XXX

4.8.6 Time computation

To perform a full order model time integration, one needs to have a model, a load and a curve describing time evolution of the load.

```
% define model and load
model=fe_time('demo bar');fe_case(model,'info')
% Define curves stack (time integration curve will be chosen later):
% - step with ones from t=0 to t=1e-3, 0 after :
model=fe_curve(model,'set','input','TestStep t1=1e-3');
% - ramp from t=.1 to t=2 with final value 1.1;
model=fe_curve(model,'set','ramp','TestRamp t0=.1 tf=2 Yf=1.1');
% - Ricker curve from t=0 to t=1e-3 with max amplitude value 1:
model=fe_curve(model,'set','ricker','TestRicker t0=0 dt=1e-3 A=1');
% - Sinus (with evaluated string depending on t time vector) :
model=fe_curve(model,'set','sinus',...
               'Test eval sin(2*pi*1000*t)');
% - Another sinus definition, explicit curve (with time vector,
%   it will be interpolated during the time integration if needed)
model=fe_curve(model,'set','sinus2',...
    struct('X',linspace(0,100,10)',...
```

```
    'Y',sin(linspace(0,100,10)')); % tabulated
% - Have load named 'Point load 1' reference 'input'
%   curve (one can choose any of the model stack
%   curve from it stack entry name) :
model=fe_case(model,'SetCurve','Point load 1','input');

cf=feplot(model) % plot the model
```

Once model is plotted in `feplot` one can edit each curve under the model properties Stack tab. Parameters can be modified. Curve can be plotted in `iiplot` using the `Show` pop-up button. One has to define the number of steps (`NStep`) and the total time to be displayed (`Tf`) and click `Using NStep & Tf`. One can also display curve on the `info TimeOpt` time options by clicking on `Using TimeOpt`.

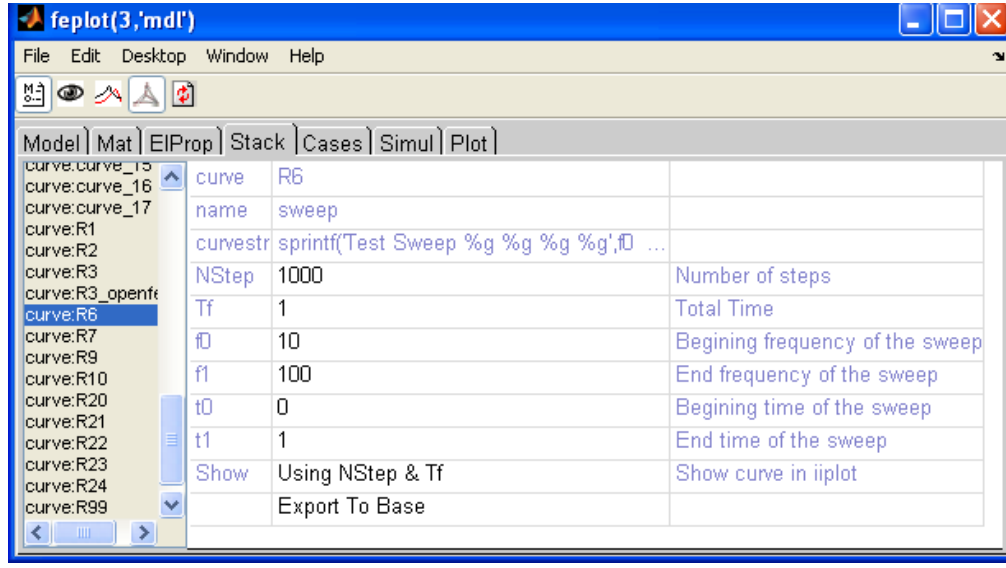


Figure 4.15: GUI associated to a curve

One can change the curve associated to the load in the Case tab.

```
% Define time computation options : dt=1e-4, 100 time steps
cf.Stack{'info','TimeOpt'}=...
    fe_time('timeopt newmark .25 .5 0 1e-4 100');
% Compute and store/display in feplot :
cf.def=fe_time(cf.mdl);
figure;plot(cf.def.data,cf.def.def(cf.def.DOF==2.01,:)); % show 2.01 result
```

Time domain responses can also be obtained by inverse transform of frequency responses as illustrated in the following example

```
model=demosdt('demo ubeam sens');DEF=fe_eig(model,[5 10 1e3]);

w=linspace(0,600,6000)';                          % define frequencies
R1=nor2xf(DEF,.001,model,w,'hz struct');          % compute freq resp.
R2=ii_mmif('ifft -struct',R1);R2.name='time';     % compute time resp.
iiplot(R2);iicom(';sub 1 1 1 1 3;ylin');          % display
```

4.8.7 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 matrix inversion (static and modal 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).

Starting with SDT 6.1, `model.Dbfile` can be defined to let SDT know that the file can be used as a database. In particular optimized assembly calls (see section 4.8.8) make use of this functionality. The database is a `.mat` file that uses the HDF5 format defined for MATLAB versions over 7.3.

For matrix inversion, the `ofact` object allows method selection. Currently the easiest to use solver (and default `ofact` method) is the multi-frontal sparse solver `spfmex`. For very large models it is recommended to use `mklserv_utils` (an implementation of IntelMKL pardiso solver), the `spfmex` solver will perform perform poorly mainly because its current implementation is not parallelized. These solvers automatically perform equation reordering so this needs not be done elsewhere. They do 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, that should be used only for very specific cases, (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` (obsolete compared to `fe_mknl`) will automatically renumber DOFs to minimize matrix bandwidth (for partial backward compatibility automatic renumbering is only done above 1000 DOF).

As SDT is an in-core oriented program, 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. It has to be noted that the most recent linux distributions (Kernel versions 4.4 and above) handle swapping quite well for large amounts of memory.

Memory management can be optimized to some extent in SDT with dedicated preferences. There is a distinction between *blockwise in-core operations*, where an intensive operation is performed by blocks to avoid large memory duplications, and *out-of-core operations* where data is written on disc to unload RAM and intensive operations involve reading file buffers and writing results buffers to temporary files. The following SDT preferences are available (they should be set by sdtdef command)

- BlasBufSize in GB, provides a block size for in-core operations, mainly matrix products with large bases, used by fe_eig, fe_norm, feutilb.
- EigOOC in GB provides a global vector basis size to trigger out-of-core operations. If a vector basis size is estimated over the specified value, it will be written to disc, used by fe_eig, fe_reduc, fe_cyclic.
- OutOfCoreBufferSize in MB provides a buffer size for out-of-core and file database operations. File database operations are common in FEMLink while handling results files. It is common not to load large files in memory. This buffer provides the amount of RAM that will still be used while in out-of-core mode, so this one should remain reasonable, and at least 10 times smaller than the EigOOC value.
- KiKeMemSize in MB provides a buffer size for out-of-core matrix assemblies, this is mostly used when exploiting FEMLink results files with matrices.
- MklServOOC a 1x2 line with [OOC_Mode MemSize(GB)]. Specific to the mklserv_utils solver with ofact allows specifying the out-of-core mode of the Pardiso solver and the associated memory threshold. OOC_Mode can take values 0 to force in-core, 2 to force out-of-core, and 1 to let the solver decide depending on MemSize. MemSize in GB is the total amount of RAM available for the solver, if the estimated factor size overcomes this value, the out-of-core mode is triggered. Beware that the solver will still need a fair amount of RAM to work, so that MemSize cannot be too small.
- MklServBufSize in GB provides a right hand size block size for in-core resolution with the mklserv_utils solver. An optimum exists around 1 GB for reasonable workstations.

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

For batch computations (in nodesktop mode) you may want to run MATLAB with the -nojvm option turned on since it increases the memory addressable by MATLAB(version ¡=6.5).

For out-of-core operations (supported by fe_mk, upcom, nasread and other functions). SDT creates temporary files whose names are generated with nas2up('tempnameExt'). You may need to set

`sdtdef('tempdir','your_dir')` to an appropriate location. The directory should be located on a local disk or a high speed disk array. If you have a RAID array or FLASH array, use a directory there.

4.8.8 Optimized assembly strategies

The handling of large models, often requires careful sequencing of assembly operations. While `fe_mknl`, `fe_load`, and `fe_case`, can be used for user defined procedures, SDT operations typically use the an internal (closed source) assembly call to `fe_case Assemble` . Illustrations of most calls can be found in `fe_simul`.

`[k,mdl,Case,Load]=fe_case(mdl,'assemble matdes 1 NoT loadback',Case);` return the stiffness without constraint elimination and evaluates loads.

`[SE,Case,Load,Sens]=fe_case(mdl,'assemble -matdes 2 1 3 4 -SE NoTload Sens')` returns desired matrices in SE.K, the associated case, load and sensors (as requested in the arguments).

Accepted command options for the assemble call are

- `-fetime` forces the nominal assembly using mass, viscous damping and stiffness, output in this order: `2 3 1`. If a reduced model is defined as an `SE,MVR`, the assembly is shortcut to output `MVR` as the assembled model, and `MVR.Case` as the Case. If the field `.Case` is absent, the case stacked in the base model is output.
- `-reset` forces reassembly even if the `.K` field is defined and filled.
- `keep` retains `model.DOF` even if some DOF are unused.
- `load` requires load assembly and output.
- `sens` requires sensor assembly and output.
- `GetT` outputs a struct containing `Case.Stack`, `Case.T` and `Case.DOF`.
- `NoT` is the usual option to prevent constraint elimination (computation of $T^T K T$). With `NoT` DOFs are given in `model.DOF` or `Case.mDOF`. Without the option they are consistent with `Case.DOF`.
- `-MatDes` specifies the list of desired matrices. Basic types are `2` for mass and `1` for stiffness, for a complete list see `MatType`.
  - `-1` is used separate matrices associated with parameters (see `upcom Par`)

  - `-1.1` removes the subparameters from the nominal matrix.

  - `-2` is used to obtain matrices associated with assembled superelements with a split based on the matrix labels (`.Klab`) only. Matrices with common labels through SE are thus assembled together. With a model having only SE, all matrices found in all SE are assembled. When the model combines SE and standard elements, the non SE elements are integrated in the first matrix of each type. To avoid this behavior specify a matrix type 1, ... where all SE and non SE elements will be assembled, then followed by SE only matrices by labels. Note that this strategy only works with a single matrix type at a time. Possibly defined matrix coefficients with a `p_super` entry are not taken into account in the SE specific matrix types.

  - `-2.1` performs the same task than `-2` but accounting for `p_super` based SE matrix coefficients.
  - `5` (geometric stiffness) uses a predefined deformation stored as stack entry `'curve','StaticState'`. Furthermore, the internal load is computed and added to returned loads.

- `InitFcn` allows pre-emptive behavior at the beginning of assembly. `ExitFcn` does the same at exit.

- `-SE` returns the assembled result as a superelement structure. One can use `-SeCDof` (superelement Case DOF) to fill `.DOF` field with constrained DOF (`Case.DOF`).

- `-cell` sets the first output as a cell array containing all assembled matrices.

- `-cfield` keeps the `Case.MatGraph` to allow further reassembly.

Structural dynamic concepts

5.1 I/O shape matrices 200
5.2 Normal mode models 202
5.3 Damping 203
5.3.1 Viscous damping in the normal mode model form 203
5.3.2 Viscous damping in finite element models 205
5.3.3 Hysteretic damping in finite element models 206
5.4 State space models 209
5.5 Complex mode models 210
5.6 Pole/residue models 212
5.7 Parametric transfer function 214
5.8 Non-parametric transfer function 214

This theoretical chapter is intended as a reference for the fundamental notions and associated variables used throughout the *SDT*. This piece of 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)

5.1 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
- 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

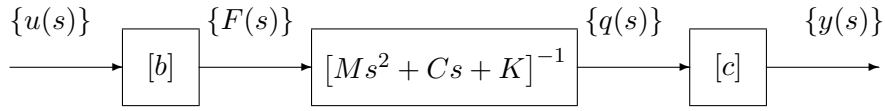
$$\begin{gathered}[Ms^2+Cs+K]_{N\times N}\{q(s)\} = [b]_{N\times NA}\{u(s)\}_{NA\times 1} \\ \{y(s)\}_{NS\times 1} = [c]_{NS\times N}\{q(s)\}_{N\times 1}\end{gathered} \tag{5.1}$$

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

$$\begin{array}{c} [M]\{q''\} + [C]\{q'\} + [K]\{q\} = [b]\{u(t)\} \\ \{y(t)\} = [c]\{q(t)\} \end{array} \tag{5.2}$$

in the time domain.

In the model form (5.1), 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



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

$$[b_l] = \begin{bmatrix} \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \end{bmatrix} \leftarrow l \tag{5.3}$$

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 292).

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 2.7.2).

- 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).

5.2 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**

$$\begin{array}{c} [[I]\, s^2 + [\Gamma]\, s + [\Omega^2]]\, \{p(s)\} = \left[\phi^T b\right] \{u(s)\} \\ \{y(s)\} = [c\phi]\, \{p(s)\} \end{array} \tag{5.4}$$

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 203).

`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 203).

`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, 2*pi for Hz). 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 4.5.3).

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).

5.3 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. [27]). This section discusses typical damping models and discusses how piece-wise Rayleigh damping is implemented in SDT.

5.3.1 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 (6.99) 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 210for 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

$$[\alpha(s)] = \sum_{j=1}^{N} \frac{\{c\phi_j\}\left\{b^T\phi_j\right\}^T}{s^2 + 2\zeta_j\omega_j s + \omega_j^2} = \sum_{j=1}^{N} \frac{[T_j]_{NS\times NA}}{s^2 + 2\zeta_j\omega_j s + \omega_j^2} \tag{5.5}$$

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

$$[Z(s)] = \left[Ms^2 + (\alpha M + \beta K)s + K\right] \tag{5.6}$$

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. The modal damping matrix thus obtained writes

$$\Gamma = \left[\backslash\alpha + \beta\omega_j^2{}_{\backslash}\right] \tag{5.7}$$

which leads to damping ratios

$$2\zeta_j = \frac{\alpha}{\omega_j} + \beta\omega_j \tag{5.8}$$

Mass coefficient α leads to high damping ratios in the low frequency range. Stiffness coefficient β leads to a damping ratio linearly increasing with the frequency.

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

$$2\zeta_j\omega_j/|\omega_j - \omega_k| \ll 1 \tag{5.9}$$

proposed by Hasselman [28], 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 un-coupling assumption can however still be applied to blocks of modes [12].

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 [12]).

5.3.2 Viscous 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 7.16 for new element creation). Viscous damping models are rarely appropriate at the finite element level [27], so that it is only supported by `celas` and `cbush` elements. Piece-wise Rayleigh damping where the viscous damping is a combination of element mass

and stiffness on element subsets

$$C = \sum_{j=1}^{NS} \left[\alpha_j^S M_j^S + \beta_j^S K_j^S \right] \tag{5.10}$$

is supported as follows. For each material or group that is to be considered in the linear combination one defines a row entry giving `GroupId MatId AlphaS BetaS` (note that some elements may be counted twice if they are related to a group and a material entry). One can alternatively define `ProId` as a 5th column (useful for celas element that have no matid). Note that each line is separately accounted for, so that duplicated entries or multiple references to same `GroupId`, `MatId` or `ProId` will also be combined. For example

```
model=demosdt('demogartfe');
model=stack_set(model,'info','Rayleigh', ...
   [10 0   1e-5  0.0; ... % Elements of group 10 (masses)
     9 0   0.0  1e-3; ... % Elements of group 9 (springs)
     0 1   0.0  1e-4;      ... % Elements with MatId 1
     0 2   0.0  1e-4]);       % Elements with MatId 2
% Note that DOF numbering may be a problem when calling 'Rayleigh'
% See sdtweb simul#feass for preferrred assembly in SDT
c=feutilb('Rayleigh',model); figure(1);spy(c);

dc=fe_ceig(model,[1 5 20 1e3]);cf=feplot(model,dc);
```

Such damping models are typically used in time integration applications. `Info,Rayleigh` entries are properly handled by `Assemble` commands.

You can also provide `model=stack_set(model,'info','Rayleigh',[alpha beta])`.

Note that in case of Rayleigh damping, celas element viscous damping will also be taken into account.

5.3.3 Hysteretic damping in finite element models

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

$$Z(s) = \left[Ms^2 + K + iB \right] = Ms^2 + \sum_{j=1}^{NE} \left[K_j^e \right] (1 + i\eta_j^e) \tag{5.11}$$

The name *loss factor* derives from the fact that η is equal to the ratio of energy dissipated for one cycle $E_d = \int_0^T \sigma\epsilon' dt$ by 2π the maximum potential energy $E_p = 1/2E$.

If dissipative materials used have a loss factor property, these are used by `Assemble` commands with a desired matrix type 4. If no material damping is defined, you can also use `DefaultZeta` to set a

global loss factor to `eta=2*DefaultZeta`.

Using complex valued constitutive parameters will not work for most element functions. Hysteretic damping models can thus be assembled using the `Rayleigh` command shown above (to assemble the imaginary part of K rather than C or using `upcom` (see section 6.4). 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 [29])

```
Up=upcom('load GartUp'); cf=feplot(Up);
Up=fe_case(Up,'parReset', ...
 'Par k','Constrained Layer','group 6', ...
 'Par 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 ];
Up=upcom(Up,'ParCoef',par);

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

% 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);
dr=fe_ceig({mr,cr,kr,[]});dr.def=def.def*dr.def;dr.DOF=def.DOF;
cf.def=dr
```

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

```
Up=fe_case(Up,'Dofload','Point loads',[4.03;55.03], ...
              'SensDof','Sensors',[4 55 30]'+.03);
Sens=feutilb('placeindof',def.DOF,fe_case(Up,'sens'));
Load=fe_load(Up);
```

```
ind=find(dr.data(:,1)>5); % flexible modes

% Standard elastic response with modal damping
f=linspace(5,60,2048);
d1=def; d1.data(7:20,2)=dr.data(ind,2);
nor2xf(d1,Up,f,'hz iiplot "Normal" -reset -po');

% Now complex modes
RES=struct('res',[],'po',dr.data(ind,:),'idopt',idopt('new'));
RES.idopt.residual=2;RES.idopt.fitting='complex';
for j1=1:length(ind); % deal with flexible modes
 Rj=(Sens.cta*dr.def(:,ind(j1))) *  ...   % c psi
     (dr.def(:,ind(j1)).'*Load.def);  % psi^T b
 RES.res(j1,:)=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
res2xf(RES,f,'hz iiplot "Res2xf"');

damp=dr.data(ind,2);
d2=def;d2.data(7:20)=sqrt(real(d2.data(7:20).^2)).*sqrt(1+i*damp*2);
nor2xf(d2,Up,f,'hz iiplot "Hysteretic"');
iicom('submagpha');
```

Note that the presence of rigid body modes, which can only be represented as residual terms in the pole/residue format (see section 5.6), 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. [30].

5.4 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

$$\begin{aligned} \{\dot{x}\} &= [A]\{x(t)\} + [B]\{u(t)\} \\ \{y\} &= [C]\{x(t)\} + [D]\{u(t)\} \end{aligned} \tag{5.12}$$

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 (5.4) is given by

$$\begin{aligned} \left\{ \begin{array}{c} p' \\ p'' \end{array} \right\} &= \left[\begin{array}{cc} 0 & I \\ -\Omega^2 & -\Gamma \end{array} \right] \left\{ \begin{array}{c} p \\ p' \end{array} \right\} + \left[\begin{array}{c} 0 \\ \phi^T b \end{array} \right] \{u(t)\} \\ \{y(t)\} &= [c\phi \;\; 0] \left\{ \begin{array}{c} p \\ p' \end{array} \right\} \end{aligned} \tag{5.13}$$

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 (5.1) is given by

$$\begin{aligned} \left\{ \begin{array}{c} q' \\ q'' \end{array} \right\} &= \left[\begin{array}{cc} 0 & I \\ -M^{-1}K & -M^{-1}C \end{array} \right] \left\{ \begin{array}{c} q \\ q' \end{array} \right\} + \left[\begin{array}{c} 0 \\ M^{-1}b \end{array} \right] \{u(t)\} \\ \{y(t)\} &= [c \;\; 0] \left\{ \begin{array}{c} q \\ q' \end{array} \right\} \end{aligned} \tag{5.14}$$

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 (5.13).

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

$$\{y(s)\} = \left([C]\,[s\;I - A]^{-1}\,[B] + [D] \right) \{u(s)\} \tag{5.15}$$

assuming that $[A]$ is diagonalizable in the basis of **complex modes**, model (5.12) is equivalent to

the diagonal model

$$\begin{array}{c}\left(s\,[I]-\left[\backslash\lambda_{j}\backslash\right]\right)\{\eta(s)\}=\left[\theta_{L}^{T}b\right]\{u\}\\ \{y\}=[c\theta_{R}]\,\{\eta(s)\}\end{array} \tag{5.16}$$

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

$$\{\theta_{jL}\}^{T}\,[A]=\lambda_{j}\,\{\theta_{jL}\}^{T}\quad\text{and}\quad[A]\,\{\theta_{jR}\}=\lambda_{j}\,\{\theta_{jR}\} \tag{5.17}$$

and verify the orthogonality conditions

$$[\theta_{L}]^{T}\,[\theta_{R}]=[I]\quad\text{and}\quad[\theta_{L}]^{T}\,[A]\,[\theta_{R}]=\left[\backslash\lambda_{j}\backslash\right] \tag{5.18}$$

The diagonal state space form corresponds to the partial fraction expansion

$$\{y(s)\}=\sum_{j=1}^{2N}\frac{\{c\psi_{j}\}\left\{\psi_{j}^{T}b\right\}}{s-\lambda_{j}}=\sum_{j=1}^{2N}\frac{[R_{j}]_{NS\times NA}}{s-\lambda_{j}} \tag{5.19}$$

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 (5.19) is heavily used for the identification routines implemented in the *SDT* (see the section on the pole/residue representation ref page 212.

5.5 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,

$$\left[M\lambda_{j}^{2}+C\lambda_{j}+K\right]\{\psi_{j}\}=\{0\} \tag{5.20}$$

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 (5.14), the following generalized state-space form is preferred

$$\begin{array}{c}\left[\begin{array}{cc}C & M\\ M & 0\end{array}\right]\left\{\begin{array}{c}q'\\ q''\end{array}\right\}+\left[\begin{array}{cc}K & 0\\ 0 & -M\end{array}\right]\left\{\begin{array}{c}q\\ q'\end{array}\right\}=\left[\begin{array}{c}b\\ 0\end{array}\right]\{u\}\\ \{y\}=\left[\begin{array}{cc}c & 0\end{array}\right]\left\{\begin{array}{c}q\\ q'\end{array}\right\}\end{array} \tag{5.21}$$

The matrices M, C and K being symmetric (assumption of reciprocity), the generalized state-space model (5.21) is symmetric. The associate left and right eigenvectors are thus equal and found by solving

$$\left(\begin{bmatrix} C & M \\ M & 0 \end{bmatrix} \lambda_j + \begin{bmatrix} K & 0 \\ 0 & -M \end{bmatrix} \right) \{\theta_j\} = \{0\} \tag{5.22}$$

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

$$\{\theta_j\} = \begin{Bmatrix} \psi_j \\ \psi_j \lambda_j \end{Bmatrix} \tag{5.23}$$

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 ($\psi_{[N+1\ldots 2N]} = \bar{\psi}_{[1\ldots N]}$ and $\lambda_{[N+1\ldots 2N]} = \bar{\lambda}_{[1\ldots N]}$).

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

$$\begin{aligned} [\theta]^T \begin{bmatrix} C & M \\ M & 0 \end{bmatrix} [\theta] &= \psi^T C \psi + \Lambda \psi^T M \psi + \psi^T M \psi \Lambda &&= \left[\backslash I \backslash \right]_{2N} \\ [\theta]^T \begin{bmatrix} K & 0 \\ 0 & -M \end{bmatrix} [\theta] &= \psi^T K \psi - \Lambda \psi^T M \psi \Lambda &&= -\left[\backslash \Lambda \backslash \right]_{2N} \end{aligned} \tag{5.24}$$

where in (5.24) 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 ([12] and section 2.8.2).

Note that with hysteretic damping (complex valued stiffness, see section 5.3.2) 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 typically not necessary so that approximations on the basis of real modes or real modes with static correction are provided. Given complex modes, you can obtain state-space models with `res2ss`. For further discussions, see Ref. [31] and low level examples in section 5.3.3 .

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 2.8.3 for more details.

5.6 Pole/residue models

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

$$[\alpha(s)] = \sum_{j=1}^{2N}\left(\frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j}\right) = \sum_{j=1}^{2N}\left(\frac{[R_j]}{s-\lambda_j}\right) \tag{5.25}$$

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 `ci.IDopt` options (see the reference section on the `idopt` function) to define the current type. `ci.IDopt.Residual` specifies which corrections are needed (the default is 3 which includes both a low and high frequency residuals). `ci.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

$$[\alpha(s)] = \sum_{j\in\texttt{identified}}\left(\frac{[R_j]}{s-\lambda_j} + \frac{[\bar{R}_j]}{s-\bar{\lambda}_j}\right) + [E] + \frac{[F]}{s^2} \tag{5.26}$$

The *SDT* always stores pole/residue models in the displacement/force format. The expression of

the force to acceleration transfer function is thus

$$[A(s)] = \sum_{j \in \texttt{identified}} \left(\frac{s^2 [R_j]}{s - \lambda_j} + \frac{s^2 [\bar{R}_j]}{s - \bar{\lambda}_j} \right) + s^2 [E] + [F] \tag{5.27}$$

The **nominal** pole/residue model above is used when `ci.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** (`ci.IDopt.Fit='Posit'`) only keep the poles with positive imaginary parts

$$[\alpha(s)] = \sum_{j \in \texttt{identified}} \left(\frac{[R_j]}{s - \lambda_j} \right) + [E] + \frac{[F]}{s^2} \tag{5.28}$$

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** (`ci.IDopt.Fit='Nor'`)

$$[\alpha(s)] = \sum_{j \in \texttt{identified}} \left(\frac{[T_j]}{s^2 + 2\zeta_j \omega_j s + \omega_j^2} \right) + [E] + \frac{[F]}{s^2} \tag{5.29}$$

can be used to identify normal modes directly under the assumption of modal damping (see damp page 203).

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

$$\texttt{res} = \begin{bmatrix} \vdots & & \cdots & \cdots & & \cdots \\ R_{j(11)} & R_{j(21)} & \cdots & R_{j(12)} & R_{j(22)} & \cdots \\ \vdots & & \ddots & \vdots & & \ddots \\ E_{11} & E_{21} & \cdots & E_{12} & E_{22} & \cdots \\ F_{11} & F_{21} & \cdots & F_{12} & F_{22} & \cdots \end{bmatrix} \tag{5.30}$$

The normal mode residues (`ci.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.

5.7 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

$$H_j(s) = \frac{a_{j,1}s^{na-1} + a_{j,2}s^{na-2} + \ldots + a_{j,na}}{b_{j,1}s^{nb-1} + b_{j,2}s^{nb-2} + \ldots + b_{j,nb}} \tag{5.31}$$

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

$$\texttt{num} = \begin{bmatrix} a_{11} & a_{12} & \ldots \\ a_{21} & a_{22} & \ldots \\ \vdots & & \ddots \end{bmatrix} \text{ and } \texttt{den} = \begin{bmatrix} b_{11} & b_{12} & \ldots \\ b_{21} & b_{22} & \ldots \\ \vdots & & \ddots \end{bmatrix} \tag{5.32}$$

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.

5.8 Non-parametric transfer function

Response data structures are the classical format to store non-parametric transfer functions. Multi-dim curve can also be used.

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)

$$\begin{Bmatrix} y_1(\omega) \\ \vdots \\ y_{NS}(\omega) \end{Bmatrix} = [H]\{u\} = \begin{bmatrix} H_{11}(\omega) & H_{12}(\omega) & \ldots \\ H_{21}(\omega) & H_{22}(\omega) & \\ \vdots & & \ddots \end{bmatrix}_{NS \times NA} \begin{Bmatrix} u_1(\omega) \\ \vdots \\ u_{NA}(\omega) \end{Bmatrix} \tag{5.33}$$

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

$$\mathtt{w} = \left\{ \begin{array}{c} \omega_1 \\ \vdots \\ \omega_{NW} \end{array} \right\}_{NW \times 1} \tag{5.34}$$

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.

$$\mathtt{xf} = \left[\begin{array}{ccccccc} H_{11}(\omega_1) & H_{21}(\omega_1) & \dots & H_{12}(\omega_1) & H_{22}(\omega_1) & \dots \\ H_{11}(\omega_2) & H_{21}(\omega_2) & \dots & H_{12}(\omega_2) & H_{22}(\omega_2) & \dots \\ \vdots & & \ddots & \vdots & & \ddots \end{array} \right]_{NW \times (NS \times NA)} \tag{5.35}$$

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

6

Advanced FEM tools

6.1 FEM problem formulations 219
6.1.1 3D elasticity 219
6.1.2 2D elasticity 220
6.1.3 Acoustics 221
6.1.4 Classical lamination theory 222
6.1.5 Piezo-electric volumes 225
6.1.6 Piezo-electric shells 227
6.1.7 Geometric non-linearity 229
6.1.8 Thermal pre-stress 231
6.1.9 Hyperelasticity 231
6.1.10 Gyroscopic effects 233
6.1.11 Centrifugal follower forces 234
6.1.12 Poroelastic materials 234
6.1.13 Heat equation 239
6.2 Model reduction theory 241
6.2.1 General framework 241
6.2.2 Normal mode models 242
6.2.3 Static correction to normal mode models 244
6.2.4 Static correction with rigid body modes 245
6.2.5 Other standard reduction bases 246
6.2.6 Substructuring 247
6.2.7 Reduction for parameterized problems 249
6.3 Superelements and CMS 250
6.3.1 Superelements in a model 250
6.3.2 SE data structure reference 251
6.3.3 An example of SE use for CMS 253
6.3.4 Obsolete superelement information 255

6.3.5 Sensors and superelements 256
6.4 Model parameterization 258
6.4.1 Parametric models, zCoef 258
6.4.2 Reduced parametric models 261
6.4.3 upcom parameterization for full order models 261
6.4.4 Getting started with upcom 262
6.4.5 Reduction for variable models 263
6.4.6 Predictions of the response using upcom 264
6.5 Finite element model updating 265
6.5.1 Error localization/parameter selection 266
6.5.2 Update based on frequencies 267
6.5.3 Update based on FRF 267
6.6 Handling models with piezoelectric materials 269
6.7 Viscoelastic modeling tools 269
6.8 SDT Rotor 269

6.1 FEM problem formulations

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

6.1.1 3D elasticity

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

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

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

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \end{Bmatrix} = \begin{bmatrix} N,x & 0 & 0 \\ 0 & N,y & 0 \\ 0 & 0 & N,z \\ 0 & N,z & N,y \\ N,z & 0 & N,x \\ N,y & N,x & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{6.1}$$

where the engineering notation $\gamma_{yz} = 2\epsilon_{yz}$, ... is used. Stress by

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sigma_{yz} \\ \sigma_{zx} \\ \sigma_{xy} \end{Bmatrix} = \begin{bmatrix} d_{1,1}N,x+d_{1,5}N,z+d_{1,6}N,y & d_{1,2}N,y+d_{1,4}N,z+d_{1,6}N,x & d_{1,3}N,z+d_{1,4}N,y+d_{1,5}N,x \\ d_{2,1}N,x+d_{2,5}N,z+d_{2,6}N,y & d_{2,2}N,y+d_{2,4}N,z+d_{2,6}N,x & d_{2,3}N,z+d_{2,4}N,y+d_{2,5}N,x \\ d_{3,1}N,x+d_{3,5}N,z+d_{3,6}N,y & d_{3,2}N,y+d_{3,4}N,z+d_{3,6}N,x & d_{3,3}N,z+d_{3,4}N,y+d_{3,5}N,x \\ d_{4,1}N,x+d_{4,5}N,z+d_{4,6}N,y & d_{4,2}N,y+d_{4,4}N,z+d_{4,6}N,x & d_{4,3}N,z+d_{4,4}N,y+d_{4,5}N,x \\ d_{5,1}N,x+d_{5,5}N,z+d_{5,6}N,y & d_{5,2}N,y+d_{5,4}N,z+d_{5,6}N,x & d_{5,3}N,z+d_{5,4}N,y+d_{5,5}N,x \\ d_{6,1}N,x+d_{6,5}N,z+d_{6,6}N,y & d_{6,2}N,y+d_{6,4}N,z+d_{6,6}N,x & d_{6,3}N,z+d_{6,4}N,y+d_{6,5}N,x \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{6.2}$$

Note that the strain states are $\{\epsilon_x \;\; \epsilon_y \;\; \epsilon_z \;\; \gamma_{yz} \;\; \gamma_{zx} \;\; \gamma_{xy}\}$ which may not be the convention of other software.

Note that NASTRAN, SAMCEF, ANSYS and MODULEF order shear stresses with $\sigma_{xy}, \sigma_{yz}, \sigma_{zx}$ (MODULEF elements are obtained by setting p_solid integ value to zero). Abaqus uses $\sigma_{xy}, \sigma_{xz}, \sigma_{yz}$

In fe_stress the stress reordering can be accounted for by the definition of the proper TensorTopology matrix.

For isotropic materials

$$D = \begin{bmatrix} \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 \end{bmatrix} & 0 \\ 0 & \begin{bmatrix} G & 0 & 0 \\ 0 & G & 0 \\ 0 & 0 & G \end{bmatrix} \end{bmatrix} \tag{6.3}$$

with at nominal $G = E/(2(1+\nu))$. For isotropic materials, interpolation of $\rho, \eta, E, \nu, G, \alpha$ with temperature is supported.

For orthotropic materials, the compliance is given by

$$\{\epsilon\} = [D]^{-1}\{\sigma\} = \begin{bmatrix} 1/E_1 & -\frac{\nu_{21}}{E_2} & -\frac{\nu_{31}}{E_3} & 0 & 0 & 0 \\ -\frac{\nu_{12}}{E_1} & 1/E_2 & -\frac{\nu_{32}}{E_3} & 0 & 0 & 0 \\ -\frac{\nu_{13}}{E_1} & -\frac{\nu_{23}}{E_2} & 1/E_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{G_{23}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{G_{31}} & \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{G_{12}} \end{bmatrix} \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sigma_{yz} \\ \sigma_{zx} \\ \sigma_{xy} \end{Bmatrix} \tag{6.4}$$

For constitutive law building, see p_solid. Material orientation can be interpolated by defining v1 and v2 fields in the InfoAtNode. Interpolation of non isotropic material properties was only implemented for of_mk ¿= 1.236.

6.1.2 2D elasticity

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

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{Bmatrix} = \begin{bmatrix} N,x & 0 \\ 0 & N,y \\ N,y & N,x \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix} \tag{6.5}$$

and stress by

$$\begin{Bmatrix} \sigma\epsilon_x \\ \sigma\epsilon_y \\ \sigma\gamma_{xy} \end{Bmatrix} = \begin{bmatrix} d_{1,1}N,x + d_{1,3}N,y & d_{1,2}N,y + d_{1,3}N,x \\ d_{2,1}N,x + d_{2,3}N,y & d_{2,2}N,y + d_{2,3}N,x \\ d_{3,1}N,x + d_{3,3}N,y & d_{3,2}N,y + d_{3,3}N,x \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix} \tag{6.6}$$

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

$$D = \frac{E}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{6.7}$$

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

$$D = \frac{E(1-\nu}{(1+\nu)(1-2\nu)}\begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \tag{6.8}$$

6.1.3 Acoustics

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

$$\begin{Bmatrix} p,x \\ p,y \\ p,z \end{Bmatrix} = \begin{bmatrix} N,x \\ N,y \\ N,z \end{bmatrix} \begin{Bmatrix} p \end{Bmatrix} \tag{6.9}$$

This replaces the earlier flui4 ... elements.

The mass and stiffness matrices are given by

$$M_{ij} = \int_\Omega \frac{1}{\rho_0 C^2} \{N_i\} \{N_j\} \tag{6.10}$$

$$K_{ij} = \int_\Omega \frac{1}{\rho_0} \{N_{i,k}\} \{N_{j,k}\} \tag{6.11}$$

The source associated with a enforced velocity on a surface

$$B_i = \int_{\partial\Omega} \{N_i\} \{V_e\} \tag{6.12}$$

When an impedance $Z = \rho C R(1 + i\eta)$ is considered on a surface, the associated viscous damping

matrix is given by

$$C_{ij} = \int_{\partial\Omega_Z^e} \frac{1}{Z} \{N_i\} \{N_j\} \tag{6.13}$$

6.1.4 Classical lamination theory

Both isotropic and orthotropic materials are considered. In these cases, the general form of the 3D elastic material law is

$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \tau_{23} \\ \tau_{13} \\ \tau_{12} \end{Bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} & 0 & 0 & 0 \\ & C_{22} & C_{23} & 0 & 0 & 0 \\ & & C_{33} & 0 & 0 & 0 \\ & & & C_{44} & 0 & 0 \\ & (s) & & & C_{55} & 0 \\ & & & & & C_{66} \end{bmatrix} \begin{Bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \epsilon_{33} \\ \gamma_{23} \\ \gamma_{13} \\ \gamma_{12} \end{Bmatrix} \tag{6.14}$$

Plate formulation consists in assuming one dimension, the thickness along x_3, negligible compared with the surface dimensions. Thus, vertical stress $\sigma_{33} = 0$ on the bottom and upper faces, and assumed to be neglected throughout the thickness,

$$\sigma_{33} = 0 \Rightarrow \epsilon_{33} = -\frac{1}{C_{33}} \left(C_{13}\epsilon_{11} + C_{23}\epsilon_{22} \right), \tag{6.15}$$

and for isotropic material,

$$\sigma_{33} = 0 \Rightarrow \epsilon_{33} = -\frac{\nu}{1-\nu} \left(\epsilon_{11} + \epsilon_{22} \right). \tag{6.16}$$

By eliminating σ_{33}, the plate constitutive law is written, with engineering notations,

$$\begin{Bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{13} \end{Bmatrix} = \begin{bmatrix} Q_{11} & Q_{12} & 0 & 0 & 0 \\ Q_{12} & Q_{22} & 0 & 0 & 0 \\ 0 & 0 & Q_{66} & 0 & 0 \\ 0 & 0 & 0 & Q_{44} & 0 \\ 0 & 0 & 0 & 0 & Q_{55} \end{bmatrix} \begin{Bmatrix} \epsilon_{11} \\ \epsilon_{22} \\ \gamma_{12} \\ \gamma_{23} \\ \gamma_{13} \end{Bmatrix}. \tag{6.17}$$

The reduced stiffness coefficients Q_{ij} (i,j = 1,2,4,5,6) are related to the 3D stiffness coefficients C_{ij} by

$$Q_{ij} = \begin{cases} C_{ij} - \dfrac{C_{i3}C_{j3}}{C_{33}} & \text{if i,j=1,2,} \\ C_{ij} & \text{if i,j=4,5,6.} \end{cases} \tag{6.18}$$

The reduced elastic law for an isotropic plate becomes,

$$\left\{ \begin{array}{c} \sigma_{11} \\ \sigma_{22} \\ \tau_{12} \end{array} \right\} = \frac{E}{(1-\nu^2)} \left[\begin{array}{ccc} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{array} \right] \left\{ \begin{array}{c} \epsilon_{11} \\ \epsilon_{22} \\ \gamma_{12} \end{array} \right\}, \tag{6.19}$$

and

$$\left\{ \begin{array}{c} \tau_{23} \\ \tau_{13} \end{array} \right\} = \frac{E}{2(1+\nu)} \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right] \left\{ \begin{array}{c} \gamma_{23} \\ \gamma_{13} \end{array} \right\}. \tag{6.20}$$

Under Reissner-Mindlin's kinematic assumption the linearized strain tensor is

$$\epsilon = \left[\begin{array}{ccc} u_{1,1} + x_3\beta_{1,1} & \frac{1}{2}(u_{1,2} + u_{2,1} + x_3(\beta_{1,2} + \beta_{2,1})) & \frac{1}{2}(\beta_1 + w_{,1}) \\ & u_{2,2} + x_3\beta_{2,2} & \frac{1}{2}(\beta_2 + w_{,2}) \\ (s) & & 0 \end{array} \right]. \tag{6.21}$$

So, the strain vector is written,

$$\{\epsilon\} = \left\{ \begin{array}{c} \epsilon^m_{11} + x_3\kappa_{11} \\ \epsilon^m_{22} + x_3\kappa_{22} \\ \gamma^m_{12} + x_3\kappa_{12} \\ \gamma_{23} \\ \gamma_{13} \end{array} \right\}, \tag{6.22}$$

with ϵ^m the membrane, κ the curvature or bending, and γ the shear strains,

$$\epsilon^m = \left\{ \begin{array}{c} u_{1,1} \\ u_{2,2} \\ u_{1,2} + u_{2,1} \end{array} \right\}, \ \kappa = \left\{ \begin{array}{c} \beta_{1,1} \\ \beta_{2,2} \\ \beta_{1,2} + \beta_{2,1} \end{array} \right\}, \ \gamma = \left\{ \begin{array}{c} \beta_2 + w_{,2} \\ \beta_1 + w_{,1} \end{array} \right\}, \tag{6.23}$$

Note that the engineering notation with $\gamma_{12} = u_{1,2} + u_{2,1}$ is used here rather than the tensor notation with $\epsilon_{12} = (u_{1,2} + u_{2,1})/2$. Similarly $\kappa_{12} = \beta_{1,2} + \beta_{2,1}$, where a factor 1/2 would be needed for the tensor.

The plate formulation links the stress resultants, membrane forces $N_{\alpha\beta}$, bending moments $M_{\alpha\beta}$ and shear forces $Q_{\alpha 3}$, to the strains, membrane ϵ^m, bending κ and shearing γ,

$$\left\{ \begin{array}{c} N \\ M \\ Q \end{array} \right\} = \left[\begin{array}{ccc} A & B & 0 \\ B & D & 0 \\ 0 & 0 & F \end{array} \right] \left\{ \begin{array}{c} \epsilon^m \\ \kappa \\ \gamma \end{array} \right\}. \tag{6.24}$$

The stress resultants are obtained by integrating the stresses through the thickness of the plate,

$$N_{\alpha\beta} = \int_{hb}^{ht} \sigma_{\alpha\beta}\, dx_3, \quad M_{\alpha\beta} = \int_{hb}^{ht} x_3\, \sigma_{\alpha\beta}\, dx_3, \quad Q_{\alpha 3} = \int_{hb}^{ht} \sigma_{\alpha 3}\, dx_3, \tag{6.25}$$

with $\alpha, \beta = 1, 2$.

Therefore, the matrix extensional stiffness matrix $[A]$, extension/bending coupling matrix $[B]$, and the bending stiffness matrix $[D]$ are calculated by integration over the thickness interval $[hb \;\; ht]$

$$\begin{aligned} A_{ij} &= \int_{hb}^{ht} Q_{ij}\, dx_3, \qquad B_{ij} = \int_{hb}^{ht} x_3\, Q_{ij}\, dx_3, \\ D_{ij} &= \int_{hb}^{ht} x_3^2\, Q_{ij}\, dx_3, \qquad F_{ij} = \int_{hb}^{ht} Q_{ij}\, dx_3. \end{aligned} \tag{6.26}$$

An improvement of Mindlin's plate theory with tranverse shear consists in modifying the shear coefficients F_{ij} by

$$H_{ij} = k_{ij} F_{ij}, \tag{6.27}$$

where k_{ij} are correction factors. Reddy's 3^{rd} order theory brings to $k_{ij} = \frac{2}{3}$. Very commonly, enriched 3^{rd} order theory are used, and k_{ij} are equal to $\frac{5}{6}$ and give good results. For more details on the assessment of the correction factor, see [32].

For an isotropic symmetric plate ($hb = -ht = h/2$), the in-plane normal forces N_{11}, N_{22} and shear force N_{12} become

$$\left\{ \begin{array}{c} N_{11} \\ N_{22} \\ N_{12} \end{array} \right\} = \frac{Eh}{1-\nu^2} \left[\begin{array}{ccc} 1 & \nu & 0 \\ & 1 & 0 \\ (s) & & \frac{1-\nu}{2} \end{array} \right] \left\{ \begin{array}{c} u_{1,1} \\ u_{2,2} \\ u_{1,2} + u_{2,1} \end{array} \right\}, \tag{6.28}$$

the 2 bending moments M_{11}, M_{22} and twisting moment M_{12}

$$\left\{ \begin{array}{c} M_{11} \\ M_{22} \\ M_{12} \end{array} \right\} = \frac{Eh^3}{12(1-\nu^2)} \left[\begin{array}{ccc} 1 & \nu & 0 \\ & 1 & 0 \\ (s) & & \frac{1-\nu}{2} \end{array} \right] \left\{ \begin{array}{c} \beta_{1,1} \\ \beta_{2,2} \\ \beta_{1,2}+\beta_{2,1} \end{array} \right\}, \tag{6.29}$$

and the out-of-plane shearing forces Q_{23} and Q_{13},

$$\left\{ \begin{array}{c} Q_{23} \\ Q_{13} \end{array} \right\} = \frac{Eh}{2(1+\nu)} \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right] \left\{ \begin{array}{c} \beta_2 + w_{,2} \\ \beta_1 + w_{,1} \end{array} \right\}. \tag{6.30}$$

One can notice that because the symmetry of plate, that means the reference plane is the mid-plane of the plate ($x_3(0) = 0$) the extension/bending coupling matrix $[B]$ is equal to zero.

Using expression (6.26) for a constant Q_{ij}, one sees that for a non-zero offset, one has

$$A_{ij} = h\,[Q_{ij}] \qquad B_{ij} = x_3(0)h\,[Q_{ij}] \qquad C_{ij} = (x_3(0)^2 h + h^3/12)\,[Q_{ij}] \qquad F_{ij} = h\,[Q_{ij}] \tag{6.31}$$

where is clearly appears that the constitutive matrix is a polynomial function of h, h^3, $x_3(0)^2 h$ and $x_3(0)h$. If the ply thickness is kept constant, the constitutive law is a polynomial function of $1, x_3(0), x_3(0)^2$.

6.1.5 Piezo-electric volumes

A revised version of this information is available at `http://www.sdtools.com/pdf/piezo.pdf`. Missing PDF links will be found there.

The strain state associated with piezoelectric materials is described by the six classical mechanical strain components and the electrical field components. Following the IEEE standards on piezoelectricity and using matrix notations, S denotes the strain vector and E denotes the electric field vector (V/m) :

$$\left\{ \begin{array}{c} S \\ E \end{array} \right\} = \left\{ \begin{array}{c} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \\ E_x \\ E_y \\ E_z \end{array} \right\} = \left[\begin{array}{cccc} N,x & 0 & 0 & 0 \\ 0 & N,y & 0 & 0 \\ 0 & 0 & N,z & 0 \\ 0 & N,z & N,y & 0 \\ N,z & 0 & N,x & 0 \\ N,y & N,x & 0 & 0 \\ 0 & 0 & 0 & -N,x \\ 0 & 0 & 0 & -N,y \\ 0 & 0 & 0 & -N,z \end{array} \right] \left\{ \begin{array}{c} u \\ v \\ w \\ \phi \end{array} \right\} \tag{6.32}$$

where ϕ is the electric potential (V).

The constitutive law associated with this strain state is given by

$$\left\{ \begin{array}{c} T \\ D \end{array} \right\} = \left[\begin{array}{cc} C^E & e^T \\ e & -\varepsilon^S \end{array} \right] \left\{ \begin{array}{c} S \\ -E \end{array} \right\} \tag{6.33}$$

in which D is the electrical displacement vector (a density of charge in Cb/m^2), T is the mechanical stress vector (N/m^2). C^E is the matrix of elastic constants at zero electric field ($E = 0$, short-circuited condition, see section 6.1.1 for formulas (there C^E is noted D). Note that using $-E$ rather than E makes the constitutive law symmetric.

Alternatively, one can use the constitutive equations written in the following manner :

$$\left\{ \begin{array}{c} S \\ D \end{array} \right\} = \left[\begin{array}{cc} s^E & d^T \\ d & \varepsilon^T \end{array} \right] \left\{ \begin{array}{c} T \\ E \end{array} \right\} \tag{6.34}$$

In which s^E is the matrix of mechanical compliances, $[d]$ is the matrix of piezoelectric constants ($m/V = Cb/N$):

$$[d] = \left[\begin{array}{cccccc} d_{11} & d_{12} & d_{13} & d_{14} & d_{15} & d_{16} \\ d_{21} & d_{22} & d_{23} & d_{24} & d_{25} & d_{26} \\ d_{31} & d_{32} & d_{33} & d_{34} & d_{35} & d_{36} \end{array} \right] \tag{6.35}$$

Matrices $[e]$ and $[d]$ are related through

$$[e] = [d] \left[C^E \right] \tag{6.36}$$

Due to crystal symmetries, $[d]$ may have only a few non-zero elements.

Matrix $\left[\varepsilon^S\right]$ is the matrix of dielectric constants (permittivities) under zero strain (constant volume) given by

$$\left[\varepsilon^S\right] = \left[\begin{array}{ccc} \varepsilon^S_{11} & \varepsilon^S_{12} & \varepsilon^S_{13} \\ \varepsilon^S_{21} & \varepsilon^S_{22} & \varepsilon^S_{23} \\ \varepsilon^S_{31} & \varepsilon^S_{32} & \varepsilon^S_{33} \end{array} \right] \tag{6.37}$$

It is more usual to find the value of ε^T (Permittivity at zero stress) in the datasheet. These two values are related through the following relationship :

$$\left[\varepsilon^S\right] = \left[\varepsilon^T\right] - [d]\,[e]^T \tag{6.38}$$

For this reason, the input value for the computation should be $\left[\varepsilon^T\right]$.

Also notice that usually relative permittivities are given in datasheets:

$$\varepsilon_r = \frac{\varepsilon}{\varepsilon_0} \tag{6.39}$$

ε_0 is the permittivity of vacuum (=8.854e-12 F/m)

The most widely used piezoelectric materials are PVDF and PZT. For both of these, matrix $\left[\varepsilon^T\right]$ takes the form

$$\left[\varepsilon^T\right] = \begin{bmatrix} \varepsilon_{11}^T & 0 & 0 \\ 0 & \varepsilon_{22}^T & 0 \\ 0 & 0 & \varepsilon_{33}^T \end{bmatrix} \tag{6.40}$$

For PVDF, the matrix of piezoelectric constants is given by

$$[d] = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ d_{31} & d_{32} & d_{33} & 0 & 0 & 0 \end{bmatrix} \tag{6.41}$$

and for PZT materials :

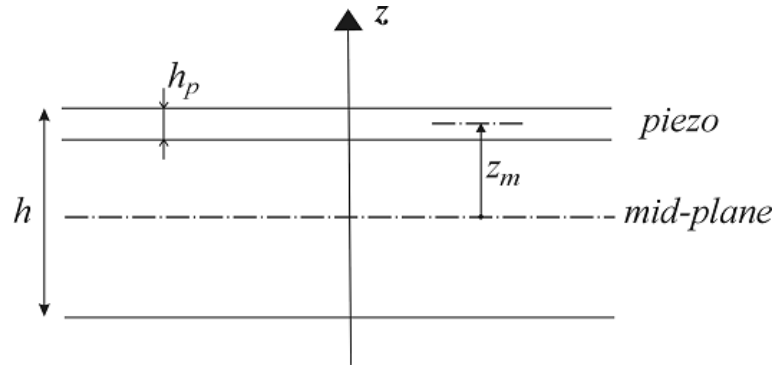
$$[d] = \begin{bmatrix} 0 & 0 & 0 & 0 & d_{15} & 0 \\ 0 & 0 & 0 & d_{24} & 0 & 0 \\ d_{31} & d_{32} & d_{33} & 0 & 0 & 0 \end{bmatrix} \tag{6.42}$$

6.1.6 Piezo-electric shells

A revised version of this information is available at `http://www.sdtools.com/pdf/piezo.pdf`.

Shell strain is defined by the membrane, curvature and transverse shear as well as the electric field components. It is assumed that in each piezoelectric layer $i = 1...n$, the electric field takes the form $\vec{E} = (0 \quad 0 \quad E_{zi})$. E_{zi} is assumed to be constant over the thickness h_i of the layer and is therefore given by $E_{zi} = -\frac{\Delta\phi_i}{h_i}$ where $\Delta\phi_i$ is the difference of potential between the electrodes at the top and bottom of the piezoelectric layer i. It is also assumed that the piezoelectric principal axes are parallel to the structural orthotropy axes.

The strain state of a piezoelectric shell takes the form



$$\begin{Bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy} \\ \kappa_{xx} \\ \kappa_{yy} \\ 2\kappa_{xy} \\ \gamma_{xz} \\ \gamma_{yz} \\ -E_{z1} \\ ... \\ -E_{zn} \end{Bmatrix} = \begin{bmatrix} N,x & 0 & 0 & 0 & 0 & 0 & ... & 0 \\ 0 & N,y & 0 & 0 & 0 & 0 & ... & 0 \\ N,y & N,x & 0 & 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & 0 & -N,x & 0 & ... & 0 \\ 0 & 0 & 0 & N,y & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & N,x & -N,y & 0 & ... & 0 \\ 0 & 0 & N,x & 0 & N & 0 & ... & 0 \\ 0 & 0 & N,y & -N & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{h_1} & ... & 0 \\ ... & ... & ... & ... & ... & 0 & ... & -\frac{1}{h_n} \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \\ ru \\ rw \\ \Delta\phi_1 \\ ... \\ \Delta\phi_n \end{Bmatrix} \tag{6.43}$$

There are thus n additional degrees of freedom $\Delta\phi_i$, n being the number of piezoelectric layers in the laminate shell

The constitutive law associated to this strain state is given by :

$$\begin{Bmatrix} N \\ M \\ Q \\ D_{z1} \\ ... \\ D_{zn} \end{Bmatrix} = \begin{bmatrix} A & B & 0 & G_1^T & ... & G_n^T \\ B & D & 0 & z_{m1}G_1^T & ... & z_{mn}G_n^T \\ 0 & 0 & F & H_1^T & ... & H_n^T \\ G_1 & z_{m1}G_1 & H_1 & -\varepsilon_1 & ... & 0 \\ ... & ... & ... & 0 & ... & 0 \\ G_n & z_{mn}G_n & H_n & 0 & ... & -\varepsilon_n \end{bmatrix} \begin{Bmatrix} \epsilon \\ \kappa \\ \gamma \\ -E_{z1} \\ ... \\ -E_{zn} \end{Bmatrix} \tag{6.44}$$

where D_{zi} is the electric displacement in piezoelectric layer (assumed constant and in the z-direction), z_{mi} is the distance between the midplane of the shell and the midplane of piezoelectric layer i, and

G_i, H_i are given by

$$G_i = \left\{ \begin{array}{ccc} e_{.1} & e_{.2} & 0 \end{array} \right\}_i [R_s]_i \tag{6.45}$$

$$H_i = \left\{ \begin{array}{cc} e_{.4} & e_{.5} \end{array} \right\}_i [R]_i \tag{6.46}$$

where . denotes the direction of polarization. If the piezoelectric is used in extension mode, the polarization is in the z-direction, therefore $H_i = 0$ and $G_i = \left\{ \begin{array}{ccc} e_{31} & e_{32} & 0 \end{array} \right\}_i$. If the piezoelectric is used in shear mode, the polarization is in the x or y-direction, therefore $G_i = 0$, and $H_i = \{0 \; e_{15}\}_i$ or $H_i = \{e_{24} \; 0\}_i$. It turns out however that the hypothesis of a uniform transverse shear strain distribution through the thickness is not satisfactory, a more elaborate shell element would be necessary. Shear actuation should therefore be used with caution.

$[R_s]_i$ and $[R]_i$ are rotation matrices associated to the angle θ of the piezoelectric layer.

$$[R_s] = \left[\begin{array}{ccc} \cos^2\theta & \sin^2\theta & \sin\theta\cos\theta \\ \sin^2\theta & \cos^2\theta & -\sin\theta\cos\theta \\ -2\sin\theta\cos\theta & 2\sin\theta\cos\theta & \cos^2\theta - \sin^2\theta \end{array} \right] \tag{6.47}$$

$$[R] = \left[\begin{array}{cc} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{array} \right] \tag{6.48}$$

6.1.7 Geometric non-linearity

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

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

$$\int_{\Omega_0} (\rho_0 u'', \delta v) + \int_{\Omega_0} S : \delta e = \int_{\Omega_0} f.\delta v \quad \forall \delta v \tag{6.49}$$

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

$$F_{i,j} = I + u_{i,j} = \delta_{ij} + \{N_{,j}\}^T \{q_i\} \tag{6.50}$$

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 really 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 used to go from engineering ($\{e_{11}\ e_{22}\ e_{33}\ 2e_{23}\ 2e_{31}\ 2e_{12}\}$) to tensor $[e_{ij}]$ notations `ind_ts_eg=[1 6 5;6 2 4;5 4 3];e_tensor=e_engineering(ind_ts_eg);`. One can also simplify a number of computations using the fact that the contraction of a symmetric and non symmetric tensor is equal to the contraction of the symmetric tensor by the symmetric part of the non symmetric tensor.

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

$$de_{ij} = \left(F^T dF\right)_{Sym} = \left(F_{ki} \left\{N_{,j}\right\}^T \{q_k\}\right)_{Sym} \tag{6.51}$$

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

$$\int_\Omega S : \delta e = \int_\Omega \left\{\delta q_k\right\}^T \left\{N_{,j}\right\} F_{ki} S_{ij} \tag{6.52}$$

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

$$K_G = \int_\Omega S_{ij} \delta u_{k,i} u_{l,j} + \int_\Omega de : \frac{\partial^2 W}{\partial e^2} : \delta e \tag{6.53}$$

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

$$K_G^e = \int_\Omega \left\{\delta q_m\right\} \left\{N_{,l}\right\} \left(F_{mk} \frac{\partial^2 W}{\partial e^2}_{ijkl} F_{ni} + S_{lj} \right) \left\{N_{,j}\right\} \{dq_n\} \tag{6.54}$$

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

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

$$S = D : e(u) = \frac{\partial^2 W}{\partial e^2} : e(u) = \lambda Tr(e) I + 2\mu e \tag{6.55}$$

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

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

$$[K(q^n)] \left\{q^{n+1} - q^n\right\} = R(q^n) = \int_\Omega f.dv - \int_{\Omega_0} S(q^n) : \delta e \tag{6.56}$$

where external forces f are assumed to be non following.

For an example see `staticNewton`.

6.1.8 Thermal pre-stress

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

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

$$S = C : (e - e_T) = \lambda Tr(e)I + 2\mu e - (C : [\alpha])(T - T_0) \tag{6.57}$$

This expression of the stress is then used in the equilibrium (6.49), the tangent matrix computation(6.53), or the Newton iteration (6.56). Note that the fixed contribution $\int_{\Omega_0}(-C : e_T) : \delta e$ can be considered as an internal load of thermal origin.

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

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

6.1.9 Hyperelasticity

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

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

$$W = C_1(J_1 - 3) + C_2(J_2 - 3) + K(J_3 - 1)^2, \tag{6.58}$$

$$W = C_1(J_1 - 3) + C_2(J_2 - 3) + K(J_3 - 1) - (C_1 + 2C_2 + K)\ln(J_3), \tag{6.59}$$

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

$$C = I + 2e, \tag{6.60}$$

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

$$J_1 = I_1 I_3^{-\frac{1}{3}}, \quad J_2 = I_2 I_3^{-\frac{2}{3}}, \quad J_3 = I_3^{\frac{1}{2}}, \tag{6.61}$$

where one recalls that

$$I_1 = \mathrm{tr}C, \quad I_2 = \frac{1}{2}\left[(\mathrm{tr}C)^2 - \mathrm{tr}C^2\right], \quad I_3 = \det C. \tag{6.62}$$

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

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

$$\frac{\partial I_1}{\partial C_{ij}} = \delta_{ij}, \quad \frac{\partial I_2}{\partial C_{ij}} = I_1\delta_{ij} - C_{ij}, \quad \frac{\partial I_3}{\partial C_{ij}} = I_3 C^{-1}_{ij}, \tag{6.63}$$

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

$$\frac{\partial^2 I_1}{\partial C_{ij}\partial C_{kl}} = 0, \quad \frac{\partial^2 I_2}{\partial C_{ij}\partial C_{kl}} = -\delta_{ik}\delta_{jl} + \delta_{ij}\delta_{kl}, \quad \frac{\partial^2 I_3}{\partial C_{ij}\partial C_{kl}} = C_{mn}\epsilon_{ikm}\epsilon_{jln}, \tag{6.64}$$

where the ϵ_{ijk} coefficients are defined by

$$\begin{cases} \epsilon_{ijk} & = 0 \quad \text{when 2 indices coincide} \\ & = 1 \quad \text{when } (i,j,k) \text{ even permutation of } (1,2,3) \\ & = -1 \quad \text{when } (i,j,k) \text{ odd permutation of } (1,2,3) \end{cases} \tag{6.65}$$

Note: when the strain components are seen as a column vector ("engineering strains") in the form $(e_{11}, e_{22}, e_{33}, 2e_{23}, 2e_{31}, 2e_{12})'$, the last two terms of (6.64) thus correspond to the following 2 matrices

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/2 \end{pmatrix}, \tag{6.66}$$

$$\begin{pmatrix} 0 & C_{33} & C_{22} & -C_{23} & 0 & 0 \\ C_{33} & 0 & C_{11} & 0 & -C_{13} & 0 \\ C_{22} & C_{11} & 0 & 0 & 0 & -C_{12} \\ -C_{23} & 0 & 0 & -C_{11}/2 & C_{12}/2 & C_{13}/2 \\ 0 & -C_{13} & 0 & C_{12}/2 & -C_{22}/2 & C_{23}/2 \\ 0 & 0 & -C_{12} & C_{13}/2 & C_{23}/2 & -C_{33}/2 \end{pmatrix}. \tag{6.67}$$

We finally use chain-rule differentiation to compute

$$S = \frac{\partial W}{\partial e} = \sum_k \frac{\partial W}{\partial I_k}\frac{\partial I_k}{\partial e}, \tag{6.68}$$

$$\frac{\partial^2 W}{\partial e^2} = \sum_k \frac{\partial W}{\partial I_k}\frac{\partial^2 I_k}{\partial e^2} + \sum_k \sum_l \frac{\partial^2 W}{\partial I_k \partial I_l}\frac{\partial I_k}{\partial e}\frac{\partial I_l}{\partial e}. \tag{6.69}$$

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

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

6.1.10 Gyroscopic effects

Written by Arnaud Sternchuss ECP/MSSMat.

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

$$\frac{\partial \mathbf{x}}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} + \mathbf{\Omega} \wedge (\mathbf{p} + \mathbf{u}) \tag{6.70}$$

and its acceleration is

$$\frac{\partial^2 \mathbf{x}}{\partial t^2} = \frac{\partial^2 \mathbf{u}}{\partial t^2} + \frac{\partial \mathbf{\Omega}}{\partial t} \wedge (\mathbf{p} + \mathbf{u}) + 2\mathbf{\Omega} \wedge \frac{\partial \mathbf{u}}{\partial \mathbf{t}} + \mathbf{\Omega} \wedge \mathbf{\Omega} \wedge (\mathbf{p} + \mathbf{u}) \tag{6.71}$$

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

$$\mathbf{\Omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{6.72}$$

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

$$[\Omega] = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{6.73}$$

The speed can be rewritten

$$\frac{\partial \mathbf{x}}{\partial t} = \frac{\partial \mathbf{u}}{\partial t} + [\Omega] (\mathbf{p} + \mathbf{u}) \tag{6.74}$$

and the acceleration becomes

$$\frac{\partial^2 \mathbf{x}}{\partial t^2} = \frac{\partial^2 \mathbf{u}}{\partial t^2} + \frac{\partial [\Omega]}{\partial t}(\mathbf{p} + \mathbf{u}) + 2 [\Omega] \frac{\partial \mathbf{u}}{\partial t} + [\Omega]^2 (\mathbf{p} + \mathbf{u}) \tag{6.75}$$

In this expression appear

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

$\mathcal{S}_0^e$ is an element of the mesh of the initial configuration $\mathcal{S}_0$ whose density is ρ_0. $[N]$ is the matrix of

shape functions on these elements, one defines the following elementary matrices

$$\begin{aligned} \left[D_g^e\right] &= \int_{\mathcal{S}_0^e} 2\rho_0 \left[N\right]^\top \left[\Omega\right] \left[N\right] \, d\mathcal{S}_0^e \quad \textit{gyroscopic coupling} \\ \left[K_a^e\right] &= \int_{\mathcal{S}_0^e} \rho_0 \left[N\right]^\top \frac{\partial[\Omega]}{\partial t} \left[N\right] \, d\mathcal{S}_0^e \quad \textit{Coriolis acceleration} \\ \left[K_g^e\right] &= \int_{\mathcal{S}_0^e} \rho_0 \left[N\right]^\top \left[\Omega\right]^2 \left[N\right] \, d\mathcal{S}_0^e \quad \textit{centrifugal softening/stiffening} \end{aligned} \tag{6.76}$$

The traditional `fe_mknl MatType` in SDT are 7 for gyroscopic coupling and 8 for centrifugal softening.

6.1.11 Centrifugal follower forces

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

$$\delta W_\omega = \int_\Omega \rho\omega^2 R(x)\delta v_R, \tag{6.77}$$

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

$$\delta v_R = n_R \cdot \delta v. \tag{6.78}$$

Thus the non-linear stiffness term is given by

$$-d\delta W_\omega = -\int_\Omega \rho\omega^2 (dR\delta v_R + Rd\delta v_R). \tag{6.79}$$

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

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

Thus, finally

$$-d\delta W_\omega = -\int_\Omega \rho\omega^2 (du_1\delta v_1 + du_2\delta v_2). \tag{6.80}$$

Which gives

$$du_1\delta v_1 + du_2\delta v_2 = \{\delta q_\alpha\}^T\{N\}\{N\}^T\{dq_\alpha\}, \tag{6.81}$$

with $\alpha = 1, 2$.

6.1.12 Poroelastic materials

The poroelastic formulation comes from [33], recalled and detailed in [34].

Domain and variables description:

Ω	Poroelastic domain	
$\partial\Omega$	Bounding surface of poroelastic domain	
n	Unit external normal of $\partial\Omega$	
u	Solid phase displacement vector	
u^F	Fluid phase displacement vector	$u^F = \frac{\phi}{\tilde{\rho}_{22}\omega^2}\nabla p - \frac{\tilde{\rho}_{12}}{\tilde{\rho}_{22}}u$
p	Fluid phase pressure	
σ	Stress tensor of solid phase	
σ^t	Total stress tensor of porous material	$\sigma^t = \sigma - \phi\left(1 + \frac{\tilde{Q}}{\tilde{R}}\right)pI$

Weak formulation, for harmonic time dependence at pulsation ω:

$$\begin{aligned}\int_\Omega \sigma(u) : \epsilon(\delta u)\ d\Omega - \omega^2 \int_\Omega \tilde{\rho}\ u.\delta u\ d\Omega - \int_\Omega \frac{\phi}{\tilde{\alpha}}\nabla p.\delta u\ d\Omega \\ - \int_\Omega \phi\left(1 + \frac{\tilde{Q}}{\tilde{R}}\right)p\nabla.\delta u\ d\Omega - \int_{\partial\Omega}(\sigma^t(u).n).\delta u\ dS = 0 \quad \forall \delta u\end{aligned} \tag{6.82}$$

$$\begin{aligned}\int_\Omega \frac{\phi^2}{\tilde{\alpha}\rho_o\omega^2}\nabla p.\nabla\delta p\ d\Omega - \int_\Omega \frac{\phi^2}{\tilde{R}}p\ \delta p\ d\Omega - \int_\Omega \frac{\phi}{\tilde{\alpha}}u.\nabla\delta p\ d\Omega \\ - \int_\Omega \phi\left(1 + \frac{\tilde{Q}}{\tilde{R}}\right)\delta p\nabla.u\ d\Omega - \int_{\partial\Omega}\phi(u^F - u).n\ \delta p\ dS = 0 \quad \forall \delta p\end{aligned} \tag{6.83}$$

Matrix formulation, for harmonic time dependence at pulsation ω:

$$\begin{bmatrix} K - \omega^2 M & -C_1 - C_2 \\ -C_1^T - C_2^T & \frac{1}{\omega^2}F - K_p \end{bmatrix} \begin{Bmatrix} u \\ p \end{Bmatrix} = \begin{Bmatrix} F_s^t \\ F_f \end{Bmatrix} \tag{6.84}$$

where the frequency-dependent matrices correspond to:

$$
\begin{array}{ll}
\displaystyle\int_\Omega \sigma(u) : \epsilon(\delta u)\ d\Omega & \Rightarrow \delta u^T K u \\
\displaystyle\int_\Omega \tilde{\rho}\ u.\delta u\ d\Omega & \Rightarrow \delta u^T M u \\
\displaystyle\int_\Omega \frac{\phi^2}{\tilde{\alpha}\rho_o} \nabla p.\nabla \delta p & \Rightarrow \delta p^T K_p p \\
\displaystyle\int_\Omega \frac{\phi^2}{\tilde{R}} p\ \delta p & \Rightarrow \delta p^T F p \\
\displaystyle\int_\Omega \frac{\phi}{\tilde{\alpha}} \nabla p.\delta u\ d\Omega & \Rightarrow \delta u^T C_1 p \\
\displaystyle\int_\Omega \phi \left(1 + \frac{\tilde{Q}}{\tilde{R}}\right) p \nabla.\delta u\ d\Omega & \Rightarrow \delta u^T C_2 p \\
\displaystyle\int_{\partial\Omega} (\sigma^t(u).n).\delta u\ dS & \Rightarrow \delta u^T F_s^t \\
\displaystyle\int_{\partial\Omega} \phi(u^F - u).n\ \delta p\ dS & \Rightarrow \delta p^T F_f
\end{array}
\tag{6.85}
$$

N.B. if the material of the solid phase is homogeneous, the frequency-dependent parameters can be eventually factorized from the matrices:

$$
\begin{bmatrix}
(1+i\eta_s)\bar{K} - \omega^2\tilde{\rho}\bar{M} & -\frac{\phi}{\tilde{\alpha}}\bar{C}_1 - \phi\left(1+\frac{\tilde{Q}}{\tilde{R}}\right)\bar{C}_2 \\
-\frac{\phi}{\tilde{\alpha}}\bar{C}_1^T - \phi\left(1+\frac{\tilde{Q}}{\tilde{R}}\right)\bar{C}_2^T & \frac{1}{\omega^2}\frac{\phi^2}{\tilde{R}}\bar{F} - \frac{\phi^2}{\tilde{\alpha}\rho_o}\bar{K}_p
\end{bmatrix}
\begin{Bmatrix} u \\ p \end{Bmatrix}
=
\begin{Bmatrix} F_s^t \\ F_f \end{Bmatrix}
\tag{6.86}
$$

where the matrices marked with bars are frequency independent:

$$
\begin{array}{lll}
K = (1+i\eta_s)\bar{K} & M = \tilde{\rho}\bar{M} & C_1 = \frac{\phi}{\tilde{\alpha}}\bar{C}_1 \\
C_2 = \phi\left(1+\frac{\tilde{Q}}{\tilde{R}}\right)\bar{C}_2 & F = \frac{\phi^2}{\tilde{R}}\bar{F} & K_p = \frac{\phi^2}{\tilde{\alpha}\rho_o}\bar{K}_p
\end{array}
\tag{6.87}
$$

Material parameters:

ϕ	Porosity of the porous material
$\bar{\sigma}$	Resistivity of the porous material
α_∞	Tortuosity of the porous material
Λ	Viscous characteristic length of the porous material
Λ'	Thermal characteristic length of the skeleton
ρ	Density of the skeleton
G	Shear modulus of the skeleton
ν	Poisson coefficient of the skeleton
η_s	Structural loss factor of the skeleton
ρ_o	Fluid density
γ	Heat capacity ratio of fluid ($= 1.4$ for air)
η	Shear viscosity of fluid ($= 1.84 \times 10^{-5}\ kg\ m^{-1}\ s^{-1}$ for air)

Constants:

$P_o = 1,01 \times 10^5\ Pa$	Ambient pressure
$Pr = 0.71$	Prandtl number

Poroelastic specific (frequency dependent) variables:

ρ_{11}	Apparent density of solid phase	$\rho_{11} = (1-\phi)\rho - \rho_{12}$
ρ_{22}	Apparent density of fluid phase	$\rho_{22} = \phi\rho_o - \rho_{12}$
ρ_{12}	Interaction apparent density	$\rho_{12} = -\phi\rho_o(\alpha_\infty - 1)$
$\tilde{\rho}$	Effective density of solid phase	$\tilde{\rho} = \tilde{\rho}_{11} - \frac{(\tilde{\rho}_{12})^2}{\tilde{\rho}_{22}}$
$\tilde{\rho}_{11}$	Effective density of solid phase	$\tilde{\rho}_{11} = \rho_{11} + \frac{\tilde{b}}{i\omega}$
$\tilde{\rho}_{22}$	Effective density of fluid phase	$\tilde{\rho}_{22} = \rho_{22} + \frac{\tilde{b}}{i\omega}$
$\tilde{\rho}_{12}$	Interaction effective density	$\tilde{\rho}_{12} = \rho_{12} - \frac{\tilde{b}}{i\omega}$
$\tilde{b}$	Viscous damping coefficient	$\tilde{b} = \phi^2\bar{\sigma}\sqrt{1 + i\frac{4\alpha_\infty^2\eta\rho_o\omega}{\bar{\sigma}^2\Lambda^2\phi^2}}$
$\tilde{\gamma}$	Coupling coefficient	$\tilde{\gamma} = \phi\left(\frac{\tilde{\rho}_{12}}{\tilde{\rho}_{22}} - \frac{\tilde{Q}}{\tilde{R}}\right)$
$\tilde{Q}$	Elastic coupling coefficient	
	Biot formulation	$\tilde{Q} = \frac{1-\phi-\frac{K_b}{K_s}}{1-\phi-\frac{K_b}{K_s}+\phi\frac{K_s}{\tilde{K}_f}}\phi K_s$
	Approximation from $K_b/K_s << 1$	$\tilde{Q} = (1-\phi)\tilde{K}_f$
$\tilde{R}$	Bulk modulus of air in fraction volume	
	Biot formulation	$\tilde{R} = \frac{\phi^2 K_s}{1-\phi-\frac{K_b}{K_s}+\phi\frac{K_s}{\tilde{K}_f}}$
	Approximation from $K_b/K_s << 1$	$\tilde{R} = \phi\tilde{K}_f$
K_b	Bulk modulus of porous material in vacuo	$K_b = \frac{2G(1+\nu)}{3(1-2\nu)}$
K_s	Bulk modulus of elastic solid	
	est. from Hashin-Shtrikman's upper bound	$K_s = \frac{1+2\phi}{1-\phi}K_b$
$\tilde{K}_f$	Effective bulk modulus of air in pores	$\tilde{K}_f = \frac{P_o}{1-\frac{\gamma-1}{\gamma\alpha'}}$
α'	Function in $\tilde{K}_f$ (Champoux-Allard model)	$\alpha' = 1 + \frac{\omega_T}{2i\omega}\left(1+\frac{i\omega}{\omega_T}\right)^{\frac{1}{2}}$
ω_T	Thermal characteristic frequency	$\omega_T = \frac{16\eta}{Pr\Lambda'^2\rho_o}$

To add here:

- coupling conditions with poroelastic medium, elastic medium, acoustic medium
- dissipated power in medium

6.1.13 Heat equation

This section is based on an OpenFEM contribution by Bourquin Frédéric and Nassiopoulos Alexandre from *Laboratoire Central des Ponts et Chaussées.*

The variational form of the Heat equation is given by

$$\int_\Omega (\rho \mathbf{c} \dot{\theta})(v)\, dx + \int_\Omega (\mathbf{K} grad\, \theta)(grad\, v)\, dx + \int_{\partial\Omega} \alpha\theta v\, d\gamma = \int_\Omega f v\, dx + \int_{\partial\Omega} (g + \alpha\theta_{ext}) v\, d\gamma \tag{6.88}$$

$$\forall v \in H^1(\Omega)$$

with

- ρ the density, c the specific heat capacity.
- $\mathbf{K}$ the conductivity tensor of the material. The tensor $\mathbf{K}$ is symmetric, positive definite, and is often taken as diagonal. If conduction is isotropic, one can write $\mathbf{K} = k(x)Id$ where $k(x)$ is called the (scalar) conductivity of the material.
- Acceptable loads and boundary conditions are
  - Internal heat source f
  - Prescribed temperature (Dirichlet condition, also called boundary condition of first kind)

    $$\theta = \theta_{ext} \quad on \quad \partial\Omega \tag{6.89}$$

    modeled using a `DofSet` case entry.
  - Prescribed heat flux g (Neumann condition, also called boundary condition of second kind)

    $$(\mathbf{K} grad\, \theta) \cdot \vec{n} = g \quad on \quad \partial\Omega \tag{6.90}$$

    leading to a load applied on the surface modeled using a `FVol` case entry.
  - Exchange and heat flux (Fourier-Robin condition, also called boundary condition of third kind)

    $$(\mathbf{K} grad\, \theta) \cdot \vec{n} + \alpha(\theta - \theta_{ext}) = g \quad on \quad \partial\Omega \tag{6.91}$$

leading to a stiffness term (modeled using a group of surface elements with stiffness proportional to α) and a load on the associated surface proportional to $g+\alpha\theta_{ext}$ (modeled using `FVol` case entries).

Test case

One considers a solid square prism of dimensions L_x, L_y, L_z in the three directions (Ox), (Oy) and (Oz) respectively. The solid is made of homogeneous isotropic material, and its conductivity tensor thus reduces to a constant k.

The faces, $\Gamma_i (i = 1..6, \cup_{i=1}^{6} \Gamma_i = \partial\Omega)$, are subject to the following boundary conditions and loads

- $f = 40$ is a constant uniform internal heat source
- $\Gamma_1\,(x = 0)$: exchange & heat flux (Fourier-Robin) given by $\alpha = 1, g_1 = \alpha\theta_{ext} + \frac{\alpha f L_x^2}{2k} = 25$
- $\Gamma_2\,(x = L_x)$: prescribed temperature : $\theta(L_x, y, z) = \theta_{ext} = 20$
- $\Gamma_3\,(y = 0)$, $\Gamma_4\,(y = L_y)$, $\Gamma_5\,(z = 0)$, $\Gamma_6\,(z = L_z)$: exchange & heat flux $g + \alpha\theta_{ext} = \alpha\theta_{ext} + \frac{\alpha f}{2k}(L_x^2 - x^2) + g_1 = 25 - \frac{x^2}{20}$

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

$$\theta(x, y, z) = -\frac{f}{2k}x^2 + \theta_{ext} + \frac{f L_x^2}{2k} = \frac{g(x)}{\alpha} = 25 - \frac{x^2}{20}$$

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

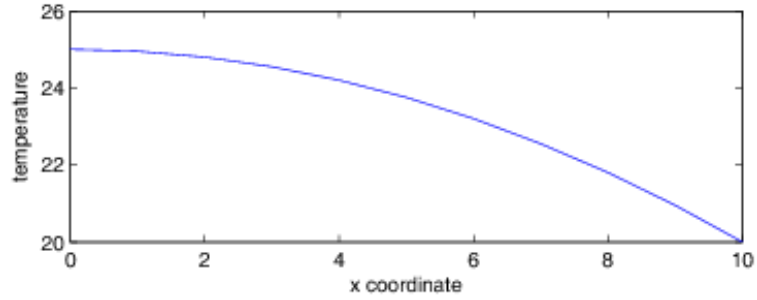


Figure 6.1: Temperature distribution along the x-axis

6.2 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 ($s = i\omega \in [\omega_1 \quad \omega_2]$)
- 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`).

6.2.1 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 (5.1) is projected as follows

$$\begin{array}{c}\left[T^T M T s^2 + T^T C T s + T^T K T\right]_{NR\times NR}\{q_R(s)\} = \left[T^T b\right]_{NR\times NA}\{u(s)\}_{NA\times 1} \\ \{y(s)\}_{NS\times 1} = [cT]_{NS\times NR}\{q_R(s)\}_{NR\times 1}\end{array} \tag{6.92}$$

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* making 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

$$[c]\,[Z(s,\alpha)]^{-1}\,[b] \approx [cT]\left[T^T Z(s,\alpha) T\right]^{-1}\left[T^T b\right] \tag{6.93}$$

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

$$[cT]\left[T^T Z T\right]^{-1}\left[T^T b\right] = \left[c\tilde{T}\right]\left[\tilde{T}^T Z \tilde{T}\right]^{-1}\left[\tilde{T}^T b\right] \tag{6.94}$$

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).

6.2.2 Normal mode models

Normal modes are defined by the eigenvalue problem

$$-[M]\,\{\phi_j\}\,\omega_j^2 + [K]_{N\times N}\,\{\phi_j\}_{N\times 1} = \{0\}_{N\times 1} \tag{6.95}$$

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 $\left[\backslash\omega_j^2\backslash\right]$).

As solutions of the eigenvalue problem (6.95), the full set of N normal modes verify two **orthogonality conditions** with respect to the mass and the stiffness

$$[\phi]^T [M] [\phi] = \left[\backslash\mu_j\backslash\right]_{N\times N} \quad \text{and} \quad [\phi]^T [K] [\phi] = \left[\backslash\mu_j\omega_j^2\backslash\right] \tag{6.96}$$

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 $\tilde{\phi}_j = \phi_j/(c_l\phi_j)$.

$$m_j(c_l) = (c_l\phi_j)^{-2} \tag{6.97}$$

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

$$k_j(c_l) = (c_l\phi_j)^{-2}\,\omega_j^2 \tag{6.98}$$

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

$$[\alpha(s)] = \sum_{j=1}^{N} \frac{\{c\phi_j\}\left\{\phi_j^T b\right\}}{s^2+\omega_j^2} \tag{6.99}$$

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 (6.99) is truncated, and a correction for the truncated terms is introduced (see section 6.2.3).

Note that the concept of effective mass [35], used for rigid base excitation tests, is very similar to the notion of generalized mass.

6.2.3 Static correction to normal mode models

Normal modes are computed to obtain the spectral decomposition (6.99). 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

$$[c]\left[Ms^2+K\right]^{-1}[b] \approx \sum_{j=1}^{N_R} \frac{[c]\{\phi_j\}\{\phi_j\}^T[b]}{s^2+\omega_j^2} + \sum_{j=N_R+1}^{N} \frac{[c]\{\phi_j\}\{\phi_j\}^T[b]}{\omega_j^2} \tag{6.100}$$

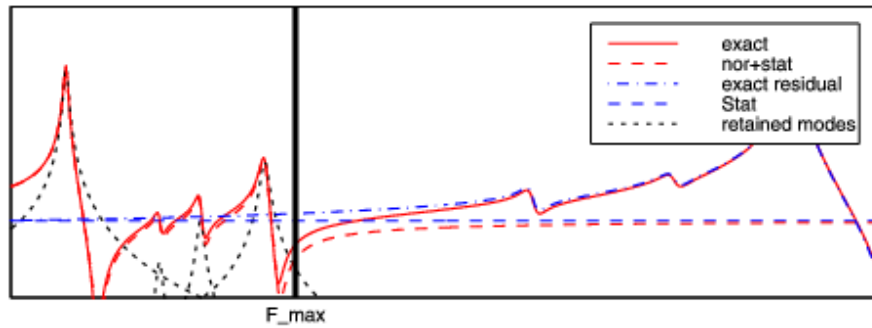


Figure 6.2: Normal mode corrections.

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 212).

For applications in reduction of finite element models, a little more work is typically done. From the

orthogonality conditions (6.96), one can easily show that for a structure with no rigid body modes (modes with $\omega_j = 0$)

$$[T_A] = [K]^{-1}[b] = \sum_{j=1}^{N} \frac{\{\phi_j\}\left\{\phi_j^T b\right\}}{\omega_j^2} \tag{6.101}$$

The static responses $K^{-1}b$ are called **attachment modes** in Component Mode Synthesis applications [36]. 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

$$[T_{AR}] = [K]^{-1}[b] - \sum_{j=1}^{NR} \frac{\{\phi_j\}\left\{\phi_j^T b\right\}}{\omega_j^2} \tag{6.102}$$

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 solicitation 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.

6.2.4 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

$$[K]^*[b] = \sum_{j=NB+1}^{N} \frac{\{\phi_j\}\left\{\phi_j^T b\right\}}{\omega_j^2} \tag{6.103}$$

This corresponds to the definition of **attachment modes** for free floating structures [36]. 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 [37] or [38] 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

$$[T_{AS}] = [K + \alpha M]^{-1} [b] = \sum_{j=1}^{N} \frac{\{\phi_j\} \left\{\phi_j^T b\right\}}{(\omega_j^2 + \alpha)} \tag{6.104}$$

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*.

6.2.5 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

$$\begin{bmatrix} Z_{II}(s) & Z_{IC}(s) \\ Z_{CI}(s) & Z_{CC}(s) \end{bmatrix} \begin{Bmatrix} < q_I(s) > \\ q_C(s) \end{Bmatrix} = \begin{Bmatrix} R_I(s) \\ < 0 > \end{Bmatrix} \tag{6.105}$$

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

$$\{q(s)\} = \begin{bmatrix} I \\ -Z_{CC}^{-1} Z_{CI} \end{bmatrix} \{q_I\} \tag{6.106}$$

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** [21].

This reduction does not fulfill the requirement of validity over a given frequency range. Craig and Bampton [39] 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

$$\left\{ \begin{array}{c} q_I(s) \\ q_C(s) \end{array} \right\} = \left[\begin{array}{cc} I & 0 \\ -K_{CC}^{-1}K_{CI} & \phi_C \end{array} \right] \{q_R\} \tag{6.107}$$

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

$$[c_I]\left[\tilde{T}\right] = \left[[I]_{NI\times NI} \quad [0]_{NI\times(NR-NI)}\right] \tag{6.108}$$

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 6.3.3).

6.2.6 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. [36] details the historical perspective while this section gives the point of view driving the *SDT* architecture (see also [40]).

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.

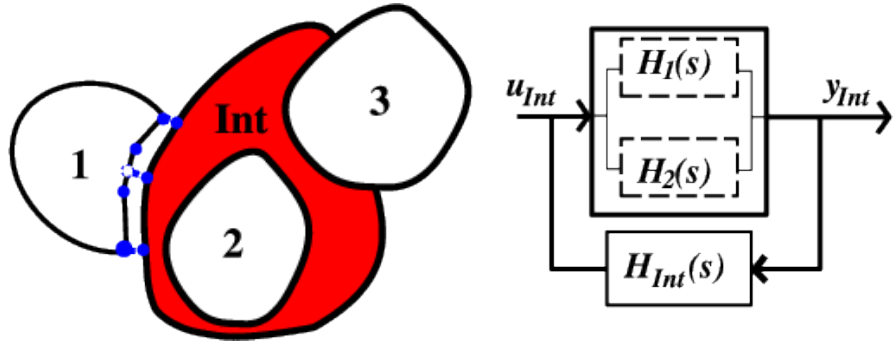


Figure 6.3: CMS procedure.

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

$$\left(\begin{bmatrix} Z_1 & 0 \\ 0 & Z_2 \end{bmatrix} + \begin{bmatrix} c_1^T & 0 \\ 0 & c_2^T \end{bmatrix} [Z_{\mathtt{int}}] \begin{bmatrix} c_1 & 0 \\ 0 & c_2 \end{bmatrix}\right) \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = [b]\,\{u(s)\} \tag{6.109}$$

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

$$\left(\begin{bmatrix} Z_1 & 0 \\ 0 & Z_2 \end{bmatrix} + \begin{bmatrix} c_1^T & 0 \\ 0 & c_2^T \end{bmatrix} \frac{\begin{bmatrix} I & -I \\ -I & I \end{bmatrix}}{\epsilon} \begin{bmatrix} c_1 & 0 \\ 0 & c_2 \end{bmatrix}\right) \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = [b]\,\{u(s)\} \tag{6.110}$$

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)

$$\begin{bmatrix} Z_1 & 0 \\ 0 & Z_2 \end{bmatrix} \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = [b]\{u(s)\} \quad \text{with} \quad [c_1 \quad -c_2] \begin{Bmatrix} q_1 \\ q_2 \end{Bmatrix} = 0 \tag{6.111}$$

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.

In particular, this allows a reduction of the number of possible interface deformations [40]. But this reduction should be done with caution to prevent locking (excessive stiffening of the interface).

6.2.7 Reduction for parameterized problems

Methods described up to now, have not taken into account the fact that in (6.93) 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 [41].

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

$$T = \left[\phi_{1\ldots NR} \quad \left[\hat{K}\right]^{-1} [b_I]\right] \tag{6.112}$$

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

$$T = [\phi_{1\ldots NR}(\alpha_1) \quad \phi_{1\ldots NR}(\alpha_2) \quad \ldots] \tag{6.113}$$

If you do not know the parameter ranges but have only a few parameters, you should consider a model combining modeshapes and modeshape sensitivities [42] (as shown in the `gartup` demo)

$$T = \left[\phi_{1\ldots NR}(\alpha_0) \quad \frac{\partial \phi_{1\ldots NR}}{\partial \alpha} \quad \ldots\right] \tag{6.114}$$

For a better discussion of the theoretical background of fixed basis reduction for variable models see Refs. [41] and [42].

6.3 Superelements and CMS

6.3.1 Superelements in a model

A superelement is a model that is included in another global model as an element. In general superelements are reduced: the response at all DOFs is described by a linear combination of shapes characterized by generalized DOFs. The use of multiple superelements to generate system predictions is called Component Mode Synthesis (CMS). For a single superelement (`SE` structure not included in a larger model) simply use `fe_reduc` calls. This section addresses superelements integrated in a model.

Starting with SDT 6, superelements are stored as `'SE'` entries in the model stack (of the form `'SE'`, *SEname*, *SEmodel*) with field detailed in section 6.3.2 . Superelements are then referenced by element rows in a group of `SE` elements in the global model. A group of superelements in the `Elt` matrix begins by the header row `[Inf abs('SE') 0]`. Each superelement is then defined by a row of the form

```
[NameCode N1 Nend BasId Elt1 EltEnd MatId ProId EltId].
```

- `NameCode` is an identifier encoding the superelement name using `fesuper('s_name')`. It is then assumed that the model stack contains an `'SE'`,*name* entry containing the model constituting the superelement. **The encoding uses `base2dec` and is limited to 8 alphabetic lower case characters and numbers**, you can use
  `NameCode = feval(fesuper('@cleanSEname'),NameCode);` to test the name compatibility.
- `[N1 Nend]` and `[Elt1 EltEnd]` are ranges of implicit `NodeId` and `EltId` of the superelement nodes and elements in the global model. That is to say that each node or element of the superelement is identified in the global model by an `Id` that can be different from the original `Id` of the superelement model stored in the stack. For more details see `Node`.
- `BasId` is the basis identifier in the `bas` field of the global model. It allows repositioning of the superelement in the global model.
- `Elt1,EltEnd` give the range of `EltId` used to identify elements constituting the superelement. These numbers are distinct from the superelement identifier itself.
- `MatId`,`ProId`,`EltId` are used to associate properties to a given superelement. Superelements support `p_super` property entries. Material information can be used for selection purposes.

The d_cms demo illustrates the Component Mode Synthesis based on a superelement element strategy. The model of this example (shown below) is composed by two stiffened plates. CMS here consists in splitting the model into two superelement plates that will be reduced, before computation of the global model modes.

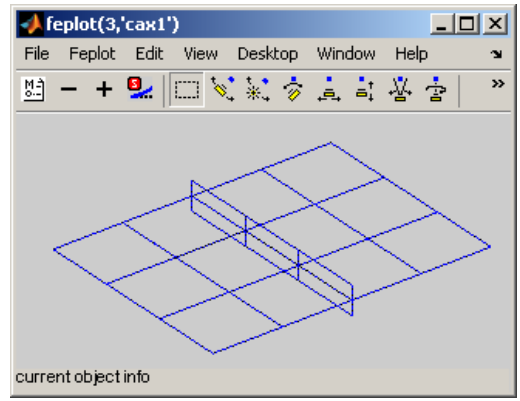


Figure 6.4: CMS example: 2 stiffened plates.

- step 1 builds the simple model shown above
- in step 2 the two parts are separated and defined as super-elements
- now display

Other examples of superelement use are given in section 6.3.3 .

6.3.2 SE data structure reference

The superelement data is stored as a 'SE',*Name*,Data entry of the global model stack. The following entries describe standard fields of the superelement Data structure (which is a standard SDT model data structure with possible additional fields).

Opt

Options characterizing the type of superelement as follows:

`Opt(1,1)`	1 classical superelements, 3 **FE update** unique superelements (see `upcom`).
`Opt(1,4)`	1 for 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 `K{i}` (where `i` is the column number). The value of `Opt(2,i)` indicates the matrix type of `K{i}`. For standard types see `MatType`.
`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. See `p_super` for high level handling.

`Node`

Nominal node matrix. Contains the nodes used by the unique superelement or the nominal generic superelement (see section 7.1). The only restriction in comparison to a standard model `Node` matrix is that it must be sorted by `NodeId` so that the last node has the largest `NodeId`.

In the element row declaring the superelement (see above) one defines a node range `N1 NEND`. The constraint on node numbers is that the defined range corresponds to the largest node number in the superelement (`NEND-N1+1=max(SE.Node(:,1))`). Not all nodes need to be defined however.

Nodes numbers in the full model are given by
`NodeId=SE.Node(:,1)-max(SE.Node(:,1))+NEND`
`N1` is really only used for coherence checking).

`K{i},Klab{i},DOF`

Superelement matrices. The presence and type of these matrices is declared in the `Opt` field (see above) and should be associated with a label giving the meaning of each matrix.

All matrices must be consistent with the `.DOF` field which is given in internal node numbering. When multiple instances of a superelement are used, node identifiers are shifted.

`Elt, Node, il, pl`

Initial model retrieval for `unique` superelements. `Elt` field contains the initial model description matrix which allows the construction of a detailed visualization as well as post-processing operations. `.Node` contains the nodes used by this model. The `.pl` and `.il` fields store material and element properties for the initial model.

Once the matrices built, `SE.Elt` may be replaced by a display mesh if appropriate.

TR

TR field contains the definition of a possible projection on a reduction basis. This information is stored in a structure array with fields

- .DOF is the model active DOF vector.
- .def is the projection matrix. There is as many columns as DOFs in the reduced basis (stored in the DOF field of the superelement structure array), and as many row as active DOFs (stored in TR.DOF).
- .adof, when appropriate, gives a list of DOF labels associated with columns of TR.def
- .data, when appropriate, gives a list frequencies associated with columns of TR.def
- .LargeDOF can be used to specify DOFs used to track the large rotation of frame where the superelement is defined in multi-body systems.
- .KeptDOF can be used to specify master DOFs not included TR.def but that should still be used for display of the superelement.

6.3.3 An example of SE use for CMS

Following example splits the 2 stiffened plane models into 2 sub models, and defines a new model with those 2 sub models taken as superelements.
First the 2 sub models are built

```
model=demosdt('Tuto CMSSE -s1 model');
SE1.Node=model.Node; SE2.Node=model.Node;
[ind,SE1.Elt]=feutil('FindElt WithNode{x>0|z>0}',model); % sel 1st plate
SE1.Node=feutil('OptimModel',SE1); SE1=feutil('renumber',SE1);
[ind,SE2.Elt]=feutil('FindElt WithNode{x<0|z<0}',model); % sel 2nd plate
SE2.Node=feutil('OptimModel',SE2); SE2=feutil('renumber',SE2);
```

Then mSE model is built including those 2 models as superelements

```
mSE.Node=[];
mSE.Elt=[Inf abs('SE') 0 0 0 0 0 0; % header row for superelements
  fesuper('s_se1') 1 16 0 1 1 100 100 1 ;    % SE1
  fesuper('s_se2') 101 116 0 2 2 101 101 2]; % SE2
mSE=stack_set(mSE,'SE','se1',SE1); mSE=stack_set(mSE,'SE','se2',SE2);
feplot(mSE); fecom('promodelinit')
```

This is a low level strategy. `fesuper` provides a set of commands to easily manipulate superelements. In particular the whole example above can be performed by a single call to `fesuper('SelAsSE')` command as shown in the CMS example in section 6.3.3 .

In this example one takes a full model split it into two superelements through element selections

```
model=demosdt('Tuto CMSSE -s1 model'); % get the full model
feutil('infoelt',model)
mSE=fesuper('SESelAsSE-dispatch',model, ...
  {'WithNode{x>0|z>0}';'WithNode{x<0|z<0}'});
feutil('infoelt',mSE)
[eltid,mSE.Elt]=feutil('eltidfix;',mSE);
```

Then the two superelements are stored in the stack of `mSE`. Both of them are reduced using `fe_reduc` (with command option `-SE` for superelement, and `-UseDof` in order to obtain physical DOFs) Craig-Bampton reduction. This operation creates the `.DOF` (reduced DOFs), `.K` (superelement reduced matrices) and `.TR` (reduction basis) fields in the superelement models.
Those operations can be performed with following commands (see `fesuper`)

```
mSE=fesuper(mSE,'setStack','se1','info','EigOpt',[5 20 1e3]);
mSE=fesuper(mSE,'settr','se1','CraigBampton -UseDof');
mSE=fesuper(mSE,'setStack','se2','info','EigOpt',[5 20 1e3]);
mSE=fesuper(mSE,'settr','se2','CraigBampton -UseDof');
```

This is the same as following lower level commands

```
SE1=stack_get(mSE,'SE','se1','getdata');
SE1=stack_set(SE1,'info','EigOpt',[5 50.1 1e3]);
SE1=fe_reduc('CraigBampton -SE -UseDof',SE1);
mSE=stack_set(mSE,'SE','se1',SE1);

SE2=stack_get(mSE,'SE','se2','getdata');
SE2=stack_set(SE2,'info','EigOpt',[5 50.1 1e3]);
SE2=fe_reduc('CraigBampton -SE -UseDof',SE2);
mSE=stack_set(mSE,'SE','se2',SE2);
```

Then the modes can be computed, using the reduced superelements

```
def=fe_eig(mSE,[5 20 1e3]);     % reduced model
dfull=fe_eig(model,[5 20 1e3]); % full model
```

The results of full and reduced models are very close. The frequency error for the first 20 modes is lower than 0.02 %.

`fesuper` provides a set of commands to manipulate superelements. `fesuper('SEAdd')` lets you add a superelement in a model. One can add a model as a unique superelement or repeat it with translations or rotations.

For CMS for example, one has to split a model into sub structure superelement models. It can be performed by the `fesuper SESelAsSE` command. This command can split a model into superelements defined by selections, or can build the model from sub models taken as superelements. The `fesuper SEDispatch` command dispatches the global model constraints (`FixDof`, `mpc`, `rbe3`, `DofSet` and `rigid` elements) into the related superelements and defines `DofSet` (imposed displacements) on the interface DOFs between sub structures.

6.3.4 Obsolete superelement information

The following strategy is now obsolete and should not be used even though it is still tested.

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`. 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*.

Reference to the superelements is done using element group headers of the form `[Inf abs('name')]`.

The `fesuper` user interface provides standard access to 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).

Warnings. In the commands superelement names must be followed by a space (in most other cases user interface commands are not sensitive to spaces).

- `Info` *Outputs a summary* of current properties of the superelement *Name*.
- `Load, Save 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 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` *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 calls of the form `fesuper('Set Name FieldOrCommand', 'Value')` are obsolete and replaced as follows

- `ref` field are now replaced by the definition of local bases for each instance of the superelement.
- `patch` simply replace the superelement `.Elt` field by another simplified model to be used for viewing once the matrices have been defined.
- `ki type 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 (see `MatType`).

6.3.5 Sensors and superelements

All sensors, excepted resultant sensor, are supported for superelement models. One can therefore add a sensor with the same way as for a standard model with `fe_case ('SensDof')` commands: `fe_case(model, 'SensDof [append, combine] SenType', Name, Sensor)`. `Name` contains the entry name in the stack of the set of sensors where `Sensor` will be added. `Sensor` is a structure of data, a vector, or a matrix, which describes the sensor (or sensors) to be added to `model`. Command option `append` specifies that the SensId of latter added sensors is increased if it is the same as a former sensor SensId. With `combine` command option, latter sensors take the place of former same SensId sensors. See section 4.6 for more details.

Following example defines some sensors in the last `mSE` model

```
% First two steps define model and split as two SE
mSE=demosdt('tuto CMSSE -s2 mSE');

mSE=fesuper(mSE,'setStack','se1','info','EigOpt',[5 50 1e3]);
mSE=fesuper(mSE,'settr','se1','CraigBampton -UseDof');
mSE=fesuper(mSE,'setStack','se2','info','EigOpt',[5 50 1e3]);
```

```
mSE=fesuper(mSE,'settr','se2','CraigBampton -UseDof');

Sensors={[0,0.0,0.75,0.0,0.0,1.0,0.0]; % Id,x,y,z,nx,ny,nz
 [0,10,0.0,0.0,1.0];                   % Id,NodeId,nx,ny,nz
 [29.01]};                             % DOF
for j1=1:length(Sensors);
  mSE=fe_case(mSE,'SensDof append trans','output',Sensors{j1});
end
mSE=fe_case(mSE,'SensDof append stress','output',[111,22,0.0,1.0,0.0]);
```

`fe_case('SensMatch')` command is the same as for standard models

```
mSE=fe_case(mSE,'SensMatch Radius2','output');
```

Use `fe_case('SensSE')` to build the observation matrix on the reduced basis

```
Sens=fe_case(mSE,'SensSE','output');
```

For resultant sensors, standard procedure does not work at this time. If the resultant sensor only relates to a specific superelement in the global model, it is however possible to define it. The strategy consists in defining the resultant sensor in the superelement model. Then one can build the observation matrix associated to this sensor, come back to the implicit nodes in the global model, and define a general sensor in the global model with the observation matrix. This strategy is described in following example.

One begins by defining resultant sensor in the related superelement

```
SE=stack_get(mSE,'SE','se2','GetData'); % get superelement
Sensor=struct('ID',0, ...
  'EltSel','WithNode{x<-0.5}');    % left part of the plate
Sensor.SurfSel='x==-0.5';          % middle line of the plate
Sensor.dir=[1.0 0.0 0.0];          % x direction
Sensor.type='resultant';           % type = resultant
SE=fe_case(SE,'SensDof append resultant',...
  'output',Sensor); % add resultant sensor to SE
```

Then one can build the associated observation matrix

```
SE=fe_case(SE,'SensMatch radius .6','output'); % SensMatch
Sens=fe_case(SE,'Sens','output');         % Build observation
```

Then one can convert the `SE` observation matrix to a `mSE` observation matrix, by renumbering DOF (this step is not necessary here since the use of `fesuper SESelAsSE` command assures that implicit numbering is the same as explicit numbering)

```
cEGI=feutil('findelt eltname SE:se2',mSE);
```

```
% implicit nodes of SE in mSE
i1=SE.Node(:,1)-max(SE.Node(:,1))+mSE.Elt(cEGI,3);
% renumber DOF to fit with the global model node numbers:
NNode=sparse(SE.Node(:,1),1,i1);
Sens.DOF=full(NNode(fix(Sens.DOF)))+rem(Sens.DOF,1);
```

Finally, one can add the resultant sensor as a general sensor

```
mSE=fe_case(mSE,'SensDof append general','output',Sens);
```

One can define a load from a sensor observation as following, and compute FRFs:

```
mSE=fe_case(mSE,'DofLoad SensDofSE','in','output:2') % from 2nd output sensor

def=fe_eig(mSE,[5 20 1e3]);     % reduced model
nor2xf(def,mSE,'acc iiplot'); ci=iiplot;
```

6.4 Model parameterization

6.4.1 Parametric models, zCoef

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 [41] for more details) as a group of models of the general second order form (5.1) where the matrices composing the dynamic stiffness depend on a number of *design parameters* p

$$[Z(p,s)] = \left[M(p)s^2 + C(p)s + K(p)\right] \tag{6.115}$$

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

$$[Z(p,s)] = \sum_{j=1}^{NB} \alpha_j(p)\,[Z_{j\alpha}(s)] \tag{6.116}$$

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.

par

SDT handles parametric models where various areas of the model are associated with a scalar coefficient weighting the model matrices (stiffness, mass, damping, ...). The first step is to define a set of parameters, which is used to decompose the full model matrix in a linear combination.

The elements are grouped in non overlapping sets, indexed m, and using the fact that element stiffness depend linearly on the considered moduli, one can represent the dynamic stiffness matrix of the parameterized structure as a linear combination of constant matrices

$$[Z(G_m,s)] = s^2\,[M] + \sum_m p_m\,[K_m] \tag{6.117}$$

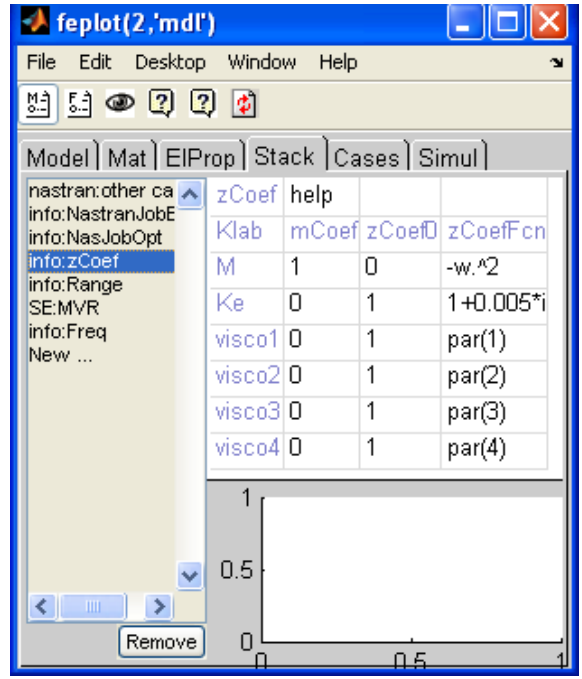
Parameters are case stack entries defined by using `fe_case par` commands (which are identical to `upcom Par` commands for an `upcom` superelement).

A parameter entry defines a element selection and a type of varying matrix. Thus

```
model=demosdt('demoubeam');
model=fe_case(model,'par k 1 .1 10','Top','withnode {z>1}');
fecom('proviewon');fecom('curtabCase','Top') % highlight the area
```

zcoef

The weighting coefficients in (6.117) are defined formally using the `cf.Stack{'info','zCoef'}` cell array viewed in the figure and detailed below.



The columns of the cell array, which can be modified with the `feplot` interface, give

- the matrix labels `Klab` which must coincide with the defined parameters
- the values of coefficients in (6.117) for the nominal mass (typically `mCoef=[1 0 0 ... ]`)
- the real valued coefficients `zCoef0` in (6.117) for the nominal stiffness K_0
- the values or strings `zCoefFcn` to be evaluated to obtain the coefficients for the dynamic stiffness (6.117).

Given a model with defined parameters/matrices, `model=fe_def('zcoef-default',model)` defines default parameters.

`zcoef=fe_def('zcoef',model)` returns weighting coefficients for a range of values using the frequencies (see `Freq`) and design point stack entries

Frequencies are stored in the model using a call of the form
`model=stack_set(model,'info','Freq',w_hertz_colum)`. Design points (temperatures, optimization points, ...) are stored as rows of the `'info','Range'` entry, see `fevisco Range` for generation.

When computing a response, `fe_def zCoef` starts by putting frequencies in a local variable `w` (which by convention is always in rd/s), and the current design point (row of `'info','Range'` entry or row of

its .val field if it exists) in a local variable par. zCoef2:end,4 is then evaluated to generate weighting coefficients zCoef giving the weighting needed to assemble the dynamic stiffness matrix (6.117). For example in a parametric analysis, where the coefficient par(1) stored in the first column of Range. One defines the ratio of current stiffness to nominal $Kvcurrent = par(1) * Kv(nominal)$ as follows

```
% external to fexf
 zCoef={'Klab','mCoef','zCoef0','zCoefFcn';
        'M'    1       0        '-w.^2';
        'Ke'   0       1        1+i*fe_def('DefEta',[]);
        'Kv'   0       1        'par(1)'};
model=struct('K',{cell(1,3)});
model=stack_set(model,'info','zCoef',zCoef);
model=stack_set(model,'info','Range', ...
   struct('val',[1;2;3],'lab',{{'par'}}));

%Within fe2xf
w=[1:10]'*2*pi;  % frequencies in rad/s
Range=stack_get(model,'info','Range','getdata');
for jPar=1:size(Range.val,1)
 Range.jPar=jPar;zCoef=fe2xf('zcoef',model,w,Range);
 disp(zCoef)
  % some work gets done here ...
end
```

6.4.2 Reduced parametric models

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

$$\begin{array}{c}\left[T^T Z(p,s) T\right]\{q_R\} = \left[T^T b\right]\{u(s)\} \\ \{y(s)\} = [cT]\{q_R\}\end{array} \tag{6.118}$$

or, using the α parameters,

$$\begin{array}{c}\sum_{j=1}^{NB} \alpha_j(p)\left[T^T \Delta Z_{j\alpha}(s) T\right]\{q_R\} = \left[T^T b\right]\{u(s)\} \\ \{y(s)\} = [cT]\{q_R\}\end{array} \tag{6.119}$$

6.4.3 upcom parameterization for full order models

Although superelements can deal with arbitrary models of the form (6.116), 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

$$[M(p)] = \sum_{j=1}^{NE} \alpha_k(p)\,[M_k^e] \qquad [K(p)] = \sum_{j=1}^{NE} \beta_k(p)\,[K_k^e] \tag{6.120}$$

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

$$\begin{aligned} &\alpha_k, \beta_k = 1 \quad \text{for} \quad k \notin e_i \\ &\alpha_k = p_i \quad \text{for} \quad k \in e_i \\ &\beta_k = p_j \quad \text{for} \quad k \in e_j \end{aligned} \tag{6.121}$$

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 Par 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)`.

6.4.4 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('ParReset')
upcom('ParAdd k','Tail','group3');
upcom('ParAdd 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');
```

6.4.5 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 6.2.7 .

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 Par commands, you must thus project the model again.

6.4.6 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 6.2.3). For a few frequency points 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).

6.5 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.

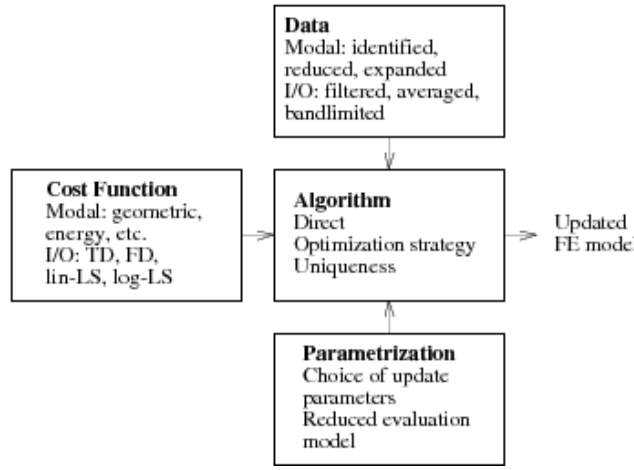


Figure 6.5: FE updating process.

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 3.2 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.

6.5.1 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 `feutil 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.

6.5.2 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 6.4.4), 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 2.2), 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 6.4.5 and start use with `upcom('opt model 1')`) can alleviate some of the difficulties but the sensitivity based method (`up_freq`) is clearly better.

6.5.3 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.

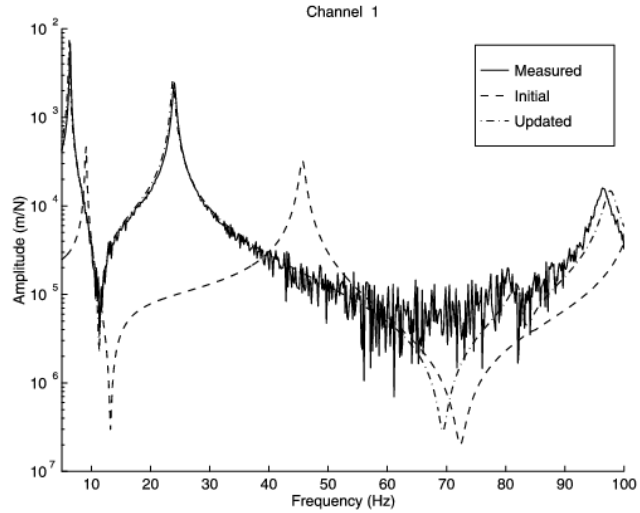


Figure 6.6: Updated FRF.

This algorithm is not very good and you are encouraged to use it as a basis for further study.

6.6 Handling models with piezoelectric materials

This has been moved to the piezoelectric manual (see `sdtweb('piezo')`) and is no longer reproduced here.

6.7 Viscoelastic modeling tools

The viscoelastic modeling tools are not part of the base SDT but licensed on an industrial basis only. Their documentation can be found at `http://www.sdtools.com/pdf/visc.pdf`.

6.8 SDT Rotor

Work on the integration of cyclic symmetry capabilities into a complete SDT ROTOR package is under progress. Their documentation can be found at `http://www.sdtools.com/pdf/rotor.pdf`.

Developer information

7.1 Nodes 273
7.1.1 Node matrix 273
7.2 Model description matrices 274
7.3 Material property matrices and stack entries 276
7.4 Element property matrices and stack entries 277
7.5 DOF definition vector 278
7.6 FEM model structure 280
7.7 FEM stack and case entries 281
7.8 FEM result data structure 285
7.9 Curves and data sets 286
7.10 DOF selection 292
7.11 Node selection 293
7.12 Element selection 296
7.13 Defining fields trough tables, expressions, ... 299
7.14 Constraint and fixed boundary condition handling 301
7.14.1 Theory and basic example 301
7.14.2 Local coordinates 302
7.14.3 Enforced displacement 303
7.14.4 Resolution as MPC and penalization transformation 303
7.14.5 Low level examples 304
7.15 Internal data structure reference 305
7.15.1 Element functions and C functionality 305
7.15.2 Standard names in assembly routines 306
7.15.3 Case.GroupInfo cell array 308
7.15.4 Element constants data structure 309
7.16 Creating new elements (advanced tutorial) 311
7.16.1 Generic compiled linear and non-linear elements 311

7.16.2 What is done in the element function 312
7.16.3 What is done in the property function 313
7.16.4 Compiled element families in of_mk 315
7.16.5 Non-linear iterations, what is done in of_mk 320
7.16.6 Element function command reference 321
7.17 Variable names and programming rules (syntax) 327
7.17.1 Variable naming conventions 328
7.17.2 Coding style 329
7.17.3 Input parsing conventions 331
7.17.4 Commands associated to project application functions 332
7.17.5 Commands associated to tutorials 336
7.18 Criteria with CritFcn 338
7.19 Legacy information 338
7.19.1 Legacy 2D elements 339
7.19.2 Rules for elements in of_mk_subs 339

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

7.1.1 Node matrix

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

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

where `NodeId` are node numbers (positive integers with no constraint on order or continuity), `PID` and `DID` are coordinate system numbers for position and displacement respectively (zero or any positive integer), `GID` is a node group number (zero or any positive integer), and `x y z` are the coordinates . For cylindrical coordinate systems, coordinates represent `r teta z` (radius, angle in degrees, and z axis value). For spherical coordinates systems, they represent `r teta phi` (radius, angle from vertical axis in degrees, azimuth in degrees). For local coordinate system support see section 7.1.1 .

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 4.5) and the mesh handling utility `femesh`.

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

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

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

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

Coordinate system handling

Local coordinate systems are stored in a `model.bas` field (see `NodeBas`). 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 3.1.1 where a local basis is defined for test results.

`feplot`, `fe_mk`, `rigid`, ... now support local coordinates. `feutil` does when the model is described 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`.

To obtain, the local to global transformation matrix (meaning $\{q_{global}\} = [c_{GL}]\{q_{local}\}$) use

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

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 4.5).

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 multi-platform 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 9 (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 7.16 .

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

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

The general format for **element rows** is

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

where

- NodeNumbers are positive integers which must match a unique NodeId identifier in the first column of the node matrix.
- MatId and ProId are material and element property identification numbers. They should be positive integers matching a unique identifier in the first column of the material pl and element il property declaration matrices.
- EltId are positive integers uniquely identifying each element. See feutil EltId for a way to return the vector and verify/fix identifiers.

- `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 and stack entries

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 4.5.1 .

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

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

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

`.name`	Description of material.
`.pl`	a single value giving the `MatId` of the corresponding row in the `model.pl` matrix or row of values. Resolution of the true `.pl` value is done by `pl=fe_mat('getpl',model)`. The property value in `.pl` should be `-1` for interpolation in `GetPl`, `-2` for interpolation using the table at each integration point, `-3` for direct use of a `FieldAtNode` value as constitutive value.
`.unit`	a two character string describing the unit system (see `fe_mat Convert` and `Unit` commands).
`.type`	the name of the material function handling this particular type of material (for example `m_elastic`).
`.field`	can be a structure allowing the interpolation of a value called *field* based on the given table. Thus `mat.E=struct('X',[-10;20],'Xlab',{{'T'}},'Y',[10 20]*1e6)` will interpolate value E based on field `T`. The positions of interpolated variables within the `pl` row are given by `list=feval(mat.type, 'propertyunittype cell', subtype)`.

7.4 Element property matrices and stack entries

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 4.5.1 .

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

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

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

`.name`	description of property.
`.il`	a single value giving the `ProId` of the corresponding row in the `il` matrix or row of values Resolution of the true `.il` value is done by `il=fe_mat('getil',model)`. The property value in `.il` should be `-1` for interpolation in `GetIl`, `-2` for interpolation using the table at each integration point, `-3` for direct use of a `FieldAtNode` value as constitutive value.
`.unit`	a two character string describing the unit system (see the `fe_mat Convert` and `Unit` commands)
`.type`	the name of the property function handling this particular type of element properties (for example `p_beam`)
`.NLdata`	used to stored non-linear property information. See `nl_spring`.
`.MAP`	specifications of a field at node, see section 7.13
`.gstate`	specifications of a field at integration points, see section 7.13
`.field`	can be a structure allowing the interpolation of a value called *field* based on the given table. Thus `pro.A=struct('X',[-10;20],'Xlab',{{'x'}},'Y',[10 20]*1e6)` will interpolate value A based on field `x`. The positions of interpolated variables within the `il` row are given by `list=feval(pro.type, 'propertyunittype cell', subtype)`.

The handling of a particular type of constants should be fully contained in the `p_*` function. The meaning of various constants should be defined in the help and TEX documentation. The subtype mechanism can be used to define several behaviors of the same class. The generation of the `integ` and `constit` vectors should be performed through a `BuildConstit` call that is the same for a full family of element shapes. The generation of `EltConst` should similarly be identical for an element family.

7.5 DOF definition vector

The meaning of each Degree of Freedom (DOF) is handled through DOF definition vectors typically stored in `.DOF` fields (and columns of `.dof` in test cases where a DOF specifies an input/output location). All informations defined at DOFs (deformations, matrices, ...) should always be stored with the corresponding DOF definition vector. The `fe_c` function supports all standard DOF manipulations (extraction, conversion to label, ...)

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

$$\texttt{mdof} = \begin{bmatrix} 1.01 \\ 1.02 \\ 1.03 \\ 1.04 \\ 1.05 \\ 1.06 \\ \vdots \end{bmatrix}, \texttt{q} = \begin{bmatrix} u_1 & u_2 & \\ v_1 & v_2 & \\ w_1 & w_2 & \\ \theta_{u1} & \theta_{u2} & \cdots \\ \theta_{v1} & \theta_{v2} & \\ \theta_{w1} & \theta_{w2} & \\ \vdots & & \ddots \end{bmatrix} \text{ and } \texttt{F} = \begin{bmatrix} F_{u1} & F_{u2} & \\ F_{v1} & F_{v2} & \\ F_{w1} & F_{w2} & \\ M_{u1} & M_{u2} & \cdots \\ M_{v1} & M_{v2} & \\ M_{w1} & M_{w2} & \\ \vdots & & \ddots \end{bmatrix} \tag{7.1}$$

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 200).
- **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 Matlab 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 6.3). 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 4.5.3 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,:). Should be associated with a .Klab field giving a string definition of each matrix. 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 and the next section for a list of standardized entries.
.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). Specifying this field let you perform conversion from materials defined in US system unit from the GUI.
.wd	working directory

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

7.7 FEM stack and case entries

Various information are stored in the model.Stack field. If you use a *SDT* handle refering to a feplot figure, modification of the model and case entries is often easier using cf.Stack calls (see feplot).

Currently supported entry types in the stack are

case	defines a case : boundary conditions, loading, ...
curve	curve to be used for simulations (see fe_curve).
info	non standard information used by solvers or meshing procedures (see below).
info,map	used to define a normal MAP, see feutil GetNormal for format
mat	defines a material entry.
pro	defines an element property entry.
SE	defines a superelement entry.
sel	defines a 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 that is a structure with fields

- .ID (a reference number of the set),
- .data defines the data
- .type nature of the set.

The following set types are acceped:

NodeId data is a column of node numbers.

EltId data is a column of element numbers.

FaceId , EdgeId data is two columns giving EltId and face/edge number (as detailed in integrules, or resulting from (tetra10('faces'), ...). Face sets are often used to define loaded surfaces.

- A third column can be added to specify subgroups within the set and a .NodeCon sparse matrix can be used to specify nodes (rows) connected to each subgroup (column).
- For FaceId sets, external code imports like used for FEMLinkface identifiers conventions may vary, so that read data may not be in coherence with SDT notations. To alleviate the problem, one can add field ConvFcn to provide a conversion function. The conversion function can be called depending on the element type ElemF with the syntax
  - feval(ConvFcn,['conv faceNum.' ElemF]); that should rethrow a renumbering vector giving in sorted SDT face numbering order the corresponding face index of the external convention.
  - feval(ConvFcn,['conv face.' ElemF]); that should rethrow the list of nodes per face (by line) in the original external face convention (but with SDT node numbering convention).

DOF values for DOF sets.

set (meta-setthe most recent set format. This structure defines a list of sub-sets gathered in a

Currently reserved names for `info` entries are

`DefaultZeta`	value to be used as default modal damping ratio (viscous damping). The default loss factor if needed is taken to be twice that value.' Default damping is only used when no other damping information is available.
`DefaultEta`	(discontinued) value to be used as default loss factor should be replaced by `DefaultZeta=eta/2`.
`EigOpt`	gives real eigenvalue solver options (see `fe_eig`).
`FluidEta`	Default loss factor for use in vibroacoustic fluid computations.
`Freq`	Frequencies given as a structure with field `.data` with frequency values and `.ID` a integer identifier. A `.unit` field can specify `rad/s`,`Hz`,`rev/mn`,`RPM`. `f=fe_def('DefFreq',model)` is used to obtain the frequency vector in Hz.
`NewNodeFrom`	integer giving the next `NodeId` to be used when adding nodes to the model (used by some commands of `feutil`).
`Omega`	rotation vector used for rotating machinery computations (see `fe_cyclic`) can be specified as a structure for unit selection. For example `r1=struct('data',250,'unit','RPM');f_hz=fe_def('deffreq',r1)`
`OrigNumbering`	original node numbering (associated with `feutil Renumber` command). Two `int32` columns giving original and new node numbers.
`StressCritFcn`	string to be evaluated for a specific stress criterion, see `fe_stress`.
`Rayleigh`	defines a Rayleigh damping entry.
`MifDes`	defines the list of desired response output (see `fe2xf`).
`NasJobOpt`	structure with options to be used for automated job runs by the NASTRAN job handler.
`NastranJobEdit`	cell array giving a list of job editing commands to be used through a `naswrite EditBulk` call.
`TimeOpt`	gives time solver options (see `fe_time`).
`TimeOptStat`	gives non-linear static solver options (see `fe_time`).

Currently reserved names for `curve` entries are

- `StaticState` used to assemble prestressed matrices (type 5).
- `q0` used to initialize time simulations and for non-linear analyses

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

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

The various cases are then stored in the `.Stack` field of the model data structure (this is done by a call to `fe_case`). If you use a *SDT* `handle` referring to a `feplot` figure, modification of the case entries is often easier using `cf.CStack` calls (see `feplot`).

7.8 FEM result data structure

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

`.def`	deformations ($NDOF$ by $NDef$ matrix)
`.DOF`	`DOF definition vector`
`.data`	(optional) ($NDef$ by N_{info} vector or matrix) characterizing the content of each deformation (frequency, time step, ...)
`.Xlab`	(optional) `{'DOF','Freq';'Index'}` cell array describing the columns of data.
`.defL`	displacement field corresponding to the left eigenmodes obtained from `fe_ceig`.
`.fun`	function description `[Model Analysis Field FieldType Format NDV]`. This is based on the UNV 55 format detailed below. Typically field with `[0 fe_curve('TypeAnalysis')]`. This field is needed for proper automated display setup.
`.lab`	(optional) cell array of strings characterizing the content of each deformation (columns of `.def`). For large arrays, the use of a `.LabFcn` is preferable.
`.ImWrite`	(optional) can be used to control automated multiple figure generation, see `iicom ImWrite`.
`.LabFcn`	callback for label generation see `fecom LabFcn`
`.Legend`	data for legend generation, see `fecom Legend`
`.label`	(optional) string describing the content
`.DofLab`	optional cell array of strings specifying a label for each DOF. This is used for display in `iiplot`.
`.scale`	field used by `feplot` to store scaling information.

The `.fun` field is a numeric row with values (a typical value for static responses is `def.fun=[0 1 0]`)

- `Model` (0 Unknown, 1 Structural, 2 Heat Transfer, 3 Fluid Flow)
- `Analysis` see list with `fe_curve('TypeAnalysis')`
- `Field` see list with 0: Unknown (or general SDT), 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 `fe_curve('typefield')`
- `Format` 0 default, 2 Real, 5 Complex
- `NDV` Number Of Data Values Per Node (0 for variable number)

SDT provides a number of utilities to manipulate deformation structures. In particular you should use

- `def=fe_def('subdef',def,ind)` extracts some deformations (columns of `def.def`). You can select based on the data field, for example with `ind=def.data(:,1)>100`.
- `def=fe_def('AppendDef',def,def1)` combines two sets of deformations
- `def=fe_def('SubDof',def,DOF)` extracts some DOF (rows of `def.def`). To select based on DOF indices, use `def=fe_def('SubDofInd',def,ind)`.
- `def=feutilb('placeindof',DOF,def)` is similar but `DOF` may be larger than `def.DOF`.
- `fe_def('SubDofInd-Cell',def,ind_dof,ind_def)` return clean display of deformation as a cell array.

7.9 Curves and data sets

Curves are used to specify `Inputs` (for time or frequency domain simulation) and store results from simulations. The basic formats are the `Multi-dim curve` and FEM result `def`. For experimental modal analysis, `Response data` and `Shapes at DOFs` are also used.

All these formats can be displayed using the `iiplot` interface. For extraction see `fe_def SubCh`.

Multi-dim curve

A curve is a `data` structure with fields

`.X` axis data. A cell array with as many entries as dimensions of `.Y`. Contents of each cell can be

- a vector (for example vector of frequencies or time steps),
- a matrix with as many rows a steps in `curve.Y`. Each column then corresponds to a different definition of the same data (time and position for example) and you can have as many rows in `curve.Xlab{i}` as colums.
- a cell array describing data vectors in `.Y` (for example response labels) with as many rows as elements in corresponding dimension of `.Y`. In such a cell array, column 2 is for units and 3 for unit type (see `fe_curve datatype`). To use a specific `curve.X{i}` to generate labels for the data, specify the index of the associated dimension in `curve.Ylab`.

.X giving x-axis data as a vector is obsolete and should be avoided.

`.Xlab` a cell array giving the meaning of each entry in `.X`. Each cell can be a string (giving the dimension name) or itself a cell array with columns giving `{'name','UnitString',unitcode,'fmt'}`. Typical entries are obtained using the `fe_curve datatypecell` command. Multiple rows can be used to describe multiple columns in the `.X` entry (matrix input for `curve.X{i}`).

`fmt`, if provided, gives a formatting instruction for example `'length=%im'`. If more intricate formatting is needed a callback can be obtained with `\zs{'#st3{'}}=sprintf(''PK=%.2fkm'',r2(j2)*1e-3);'`.

`.Y` response data with as many dimensions as the length of `curve.X` and `curve.Xlab`. If a 2D matrix rows correspond to `.X{1}` values and columns are called *channels* described by `.X{2}`.

`.Ylab` describes content of `.Y` data. It can be a string, a 1x3 unit type cell array, or a number that indicates which dimension (index in `.X{i}` field cell array) describes the `.Y` unit.

`.ID` Optional. It can be used to generate automatically vertical lines in iiplot. See `ii_plp Call from iiplot` for more details.

`.name` name of the curve used for legend generation.

`.type` Optional. `'fe_curve'`.

`.Interp` optional interpolation method. Available interpolations are `linear`, `log` and `stair`.

`.Extrap` optional extrapolation method. Available extrapolations are `flat`, `zero` (default for `fe_load`) and `exp`.

`.PlotInfo` indications for automated plotting, see `iiplot PlotInfo`

`.DimPos` order of dimensions to be shown by `iiplot`.

The following gives a basis generation example.

```
t=linspace(0,10,100)';lab={'ux';'uy'};
C1=struct('X',{{t,lab}},'Xlab',{{'Time','DOF'}}, ...
 'Y',[sin(t) cos(t)],'name','Test');
iicom('curveinit',C1.name,C1);iicom('ch1:2');
```

FEM Result

See section 7.8 or `sdtweb('def')`.

Inputs

Inputs for time or frequency simulations are stored as entries `{'curve', Name, data}` in the model stack or in the case of inputs in the `load.curve` cell array.

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

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

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

% Define input curve structure (single input step)
% For examples see: sdtweb fe_curve#Test
model=fe_curve(model,'set','input','TestStep t1=1e-3');

% define load.curve{1} to use that input
model=fe_case(model,'setcurve','Point load 1','input');

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

It is also possible to directly define the `.curve` field associated with a load

```
model=fe_time('demo bar');fe_case(model,'info')
model=fe_case(model,'remove','fd'); % loads at both ends
data=struct('DOF',[1.01;2.01],'def',1e6*eye(2),...
            'curve',{{'test ricker dt=1e-3 A=1',...
                      'test ricker dt=2e-3 A=1'}});
model = fe_case(model,'DOFLoad','Point load 1',data);

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

Response data

Response data sets `xfstruct` correspond to groups of universal files of type `UFF58` that have the same properties (type of measurement, abscissa, units, ...). They are used for identification with `idcom` while the newer curve format is used for simulation results. They are characterized by the following fields

`.w`	abscissa values
`.xf`	response data, one column per response, see section 5.8
`.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 `ufread 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. Get label with `c.group(c.dof(:,4))` for response and `c.group(c.dof(:,5))` for excitation.
`.load`	(optional) loading patterns used in the data set

The `.w` and `.xf` fields contain the real data while other fields give more precisions on its nature.

The `.dof` field describes *DOF/channel dependent options* of a MIMO data set. The `dof` field contains one row per response/DOF with the following information (this corresponds to data in line 6 of `ufread 58` except for address)

```
[RespNodeID.RespDOFID ExciNodeID.ExciDOFID Address ...
```

RespGroupID ExciGroupID FunID LoadCase ZaxisValue]

- Standard *DOF definitions* of the form NodeID.DOFID are introduced in section 7.5 . When considering sensors in general directions (see section 4.6) the SensId should match RespNodeID.RespDOFID.
- *Addresses* are integer numbers used to identify columns of xf matrices. They typically correspond to a measurement number.
- Sensor / actuator *groups* are correspond to the group names given in the group field (this is really only supported by ufread).
- Other columns are given in the universal format specification but unused in SDT.

The idopt field is used to point to identification options used on the data set. These should point to the figure options ci.IDopt.

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 and used for geometry identification.

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 5.6 .
.dof	characteristics of individual responses (follow link for description).
.fun	function characteristics (see UFF58)
.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

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 `feutil FindNode` commands provides elaborate node selection tools. Thus `femesh('findnode x>0')` returns a vector with the node numbers of all nodes in the standard global variable `FEnode` that are such that their x coordinate is positive. These can then be used to select DOFs, as shown in the section on boundary conditions section 7.14 . Node selection tools are described in the next section.

7.11 Node selection

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

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

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

Accepted selectors are

`GID i`	selects the nodes in the node group *i* (specified in column 4 of the node matrix). Logical operators are accepted.
`Group i`	selects the nodes linked to elements of group(s) *i* in the main model. Same as `InElt{Group i}`
`Groupa i`	selects nodes linked to elements of group(s) *i* of the alternate model
`InElt{sel}`	selects nodes linked to elements of the main model that are selected by the element selection command *sel*.
`NodeId >i`	selects nodes selects nodes based relation of `NodeId` to integer *i*. The logical operator `>`, `<`, `>=`, `<=`, `~=`, or `==` can be omitted (the default is then `==`). `feutil('findnode 1 2',model)` interprets the values as `NodeId` unless three values are given (then interpreted as `x y z`). `feutil('findnode',model,IdList)` should then be used.
`NotIn{sel}`	selects nodes not linked to elements of the main model that are selected by the element selection command *sel*.
`Plane == i nx ny nz`	selects nodes on the plane containing the node number *i* and orthogonal to the vector `[nx ny nz]`. Logical operators apply to the oriented half plane. *i* can be replaced by string *o xo yo zo* specifying the origin.
`rad <=r x y z`	selects nodes based on position relative to the sphere specified by radius *r* and position *x y z* node or number *x* (if *y* and *z* are not given). The logical operator `>`, `<`, `>=`, `<=` or `==` can be omitted (the default is then `<=`).
`cyl <=r i nx ny nz z1 z2`	selects nodes based on position relative to the cylinder specified by radius *r* and axis of direction *nx ny nz* and origin the node *i* (NodeId *i* can be replaced by string *o xo yo zo*). Optional arguments *z1* and *z2* define bottom and top boundaries from origin along cylinder axis.
`betweenn1 n2`	selects nodes located between the two planes of normal directed by *n1*-*n2* and respectively passing through *n1* and *n2*.
`Setname name`	finds nodes based on a set defined in the model stack. Note that the name must not contain blanks or be given between double quotes `"name"`. Set can be a `NodeId` or even an `EltId` or `FaceId`, `EdgeId set`. `"name:con IdList"` can be used to select a subset connected to nodes in the `IdList`.
`x>a`	selects nodes such that their *x* coordinate is larger than *a*. `x y z r` (where the radius `r` is taken in the *xy* plane) and the logical operators `>`, `<`, `>=`, `<=`, `==` can be used. Expressions involving other dimensions can be used for the right hand side. For example `r>.01*z+10`.
`x y z`	selects nodes with the given position. If a component is set to `NaN` it is ignored. Thus `[0 NaN NaN]` is the same as `x==0`.

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

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

Different selectors can be chained using the following logical operations

- &, finds nodes that verify both conditions.
- |, finds nodes that verify one or both conditions.
- &~ finds nodes that verify the left condition and not the right condition (exclusion from current selection state)

Condition combinations are always evaluated from left to right (parentheses are not accepted).

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

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

Basic commands are :

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

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

Accepted selectors are

`ConnectedTo` *i*	finds elements in a group that contains the nodes *i*. This calls `feutil DivideInGroups` and thus only operates on groups of elements of a single type.
`EGID` *i*	finds elements with element group identifier *i*. Operators accepted.
`EltId` *i*	finds elements with identifiers *i* in `FEelt`. Operators accepted.
`EltInd` *i*	finds elements with indices *i* in `FEelt`. Operators accepted.
`EltName` *s*	finds elements with element name *s*. `EltName flui` will select all elements with name starting with `flui`. `EltName ~ = flui` will select all elements with name not starting with `flui`. One can select superelements from their name using `EltName SE:`*SEName*. Selection of all elements but a single SE from its name is obtained using `EltName ~ = SE:`*SEName*. Regular expressions on superelement names are accepted, one then replaces token *SEName* by the prefix `#` followed by the desired expression, *e.g.* `EltName SE:`*`#tgm*`* to select all superlement whose name starts with `tgm`.
`Facing > cos x y z`	finds topologically 2-D elements whose normal projected on the direction from the element CG to *x y z* has a value superior to *cos*. Inequality operations are accepted.
`Group` *i*	finds elements in group(s) *i*. Operators accepted.
`InNode` *i*	finds elements with all nodes in the set *i*. Nodes numbers in *i* can be replaced by a string between braces defining a node selection command. For example `feutil('FindElt withnode {y>-230 & NodeId>1000}',model)`.
`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.
`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.
`Set` *i*	finds elements in element set(s) based on the `.ID` field (see `set` stack entries). Elements belonging to any set of `ID` of value *i* will be selected.

`SetName s` finds elements in element set named *s* (see `set` stack entries).

- By default an error is thrown if the set name does not exist in stack. Use command `SafeSetName` to get empty results instead.
- By default no spaces in set names are allowed. For more complicated set-names, place the name into double quotes: `SetName "my set name with spaces"`.
- Selection by exclusion is possible with token `:exclude`. *E.g.* `SetName unused:exclude` will return all elements excluding the elements present in the set named `unused`.
- Alternative calls to more advanced sets based on connectivity are possible,
  - `SetName "name:con IdList"` can be used to select a subset connected to nodes in the `IdList` (assuming the `.NodeCon` field is defined).
  - `SetName "name:subname"` can be used to select a subset in the set by connectivity format (see `set`).

`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.

`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. The `-All` option skips the internal edge elimination step. It can be combined with option `-noUni` to keep edge duplicates between elements.
Type `g` retains inter-group edges. `m` retains inter-material edges. Type `p` retains inter-property edges. `all` retains all edges. The `MatId` for the resulting model identifies the original properties of each side of the edge. The edge number is stored in the column after `EltId`.

`SelFace type` selects the external faces (surfaces) of the currently selected elements. The face number is stored in the column after `EltId` to allow set generation. See more details under `SelEdge`. The `-All` option skips the internal face elimination step. **Warning:** the face number stored in the column after the `EltId` column interferes with the `Theta` property for shell elements (see `quad4`,`tria3`). If the selection output will be used as elements in a model, ensure that the `Theta` property is properly set for your application, see `p_shell setTheta`.

`SelFace -trim val` trims a surface selection to remove boundary elements that may overcome a sharp edge. The base application is thus to be able to select interior surfaces with robustness regarding the surface edges in a volume, where it is classical to end up with a layer of side elements. The sharp edges detection uses `feutilb SurfaceAsQuad` to whom the angle defined by *val* is passed. Sharp edge element groups exclusively containing elements with nodes on the edge of the surface are then removed from the selection.

Different selectors can be chained using the available logical operations

- `&` finds elements that verify both conditions.
- `|` finds elements that verify one or both conditions.
- `&~` finds elements that verify the left condition and not the right condition (exclusion from current selection state)

`i1=feutil('FindEltGroup 1:3 & with node 1 8',model)` for example. Condition combinations are always evaluated from left to right (parentheses are not accepted). Note that `SelEdge` and `SelFace` selectors do not output elements of the mesh but new elements of respectively 1D or 2D topology, so that some combinations may not be directly possible (*e.g.* if later combined to `Group` selector).

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

Numeric values to the command can be given as additional arguments. Thus the command above could also have been written `i1=feutil('findelt group & withnode',model,1:3,[1 8])`.

7.13 Defining fields trough tables, expressions, ...

Finite element fields are used in four main formats

- `def` field at DOFs
- `InfoAtNode` field at nodes of an element group can be built from a `pro.MAP` field which can be an `VectFromDir` structure, a structure with fields `.bas` and .EltId with `EltId=0` to define material orientations.
  `info,EltOrient` is an alternative to specify the orientation of all elements rather than associate values for each property entry. The format is a structure with field `.EltId` giving the identifiers and `.bas` giving an orientation for each element in the `basis` format. To interpolate constitutive properties as a function of temperature, ... see section 7.3 .

- gstate field at integration points of an element group (can be built from a pro.gstate field).
- a field definition structure to be transformed to the other formats using a elem0('VectFromDir') command as illustrated below.

The VectFromDir structure has fields

data.dir — a cell array specifying the value of various fields. Each cell of data.dir can give a constant value, 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 and you can use the coordinates x,y,z for your evaluation.

data.lab — cell array giving label for each field of an InfoAtNode or gstate structure.

data.DOF — a vector defining the DOF associated with each .dir entry. The transformation to a vector defined at model.DOF is done using vect=elem0('VectFromDirAtDof',model,data,model.DOF).

For example

```
% Analytical expression for a displacement field
model=femesh('testubeam');
data=struct('dir',{{'ones(size(x))','y','1*x.^3'}}, ...
   'DOF',[.01;.02;.03]);
model.DOF=feutil('GetDOF',model);
def=elem0('VectFromDirAtDof',model,data,model.DOF)

% Orientation field at nodes
data=struct('dir',{{'x./sqrt(x.^2+y.^2)','y./sqrt(x.^2+y.^2)',0}}, ...
   'lab',{{'v1x','v1y','v1z'}});
pro=struct('il',1,'type','p_solid','MAP',data);
model=stack_set(model,'pro','WithMap',pro);
C1=fe_mknl('init',model);InfoAtNode=C1.GroupInfo{7}
feplot(model);fecom('showMap','WithMap') % display map

% Material field at node
sdtweb('_eval','d_mesh.m#RVEConstitInterp')
```

7.14 Constraint and fixed boundary condition handling

7.14.1 Theory and basic example

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

$$\begin{array}{c}\left[Ms^2+Cs+K\right]\{q(s)\}=[b]\{u(s)\}\\ \{y(s)\}=[c]\{q(s)\}\\ [c_{int}]\{q(s)\}=0\end{array} \tag{7.2}$$

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

$$\text{range}([T]_{N\times(N-NC)})=\ker([c_{int}]_{NS\times N}) \tag{7.3}$$

Solving problem

$$\begin{array}{c}\left[T^TMTs^2+T^TCTs+T^TKT\right]\{q_R(s)\}=\left[T^Tb\right]\{u(s)\}\\ \{y(s)\}=[cT]\{q_R(s)\}\end{array} \tag{7.4}$$

is then strictly equivalent to solving (7.2).

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

The NoT command option controls the need to return matrices, loads, ... in the full of unconstrained DOFs $[M],\{b\}$ ... or constrained T^TMT, T^Tb in fe_mknl, fe_load, ...

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

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

% Now reassemble unconstrained matrices and verify the equality
% of projected matrices
```

```
[m,k,mdof]=fe_mknl(model,'NoT');

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

To compute resultants associated with constraint forces further details are needed. One separates active DOF q_a which will be kept and slave DOF that will be eliminated q_e so that the constraint is given by

$$[c_a \;\; c_e]_{N\times N_e}\left\{\begin{array}{c} q_a \\ q_e \end{array}\right\} = 0 \;\Leftrightarrow\; \left[-(-c_e^{-1}c_a) \;\; I\right]\left\{\begin{array}{c} q_a \\ q_e \end{array}\right\} = [-G \; I]\,\{q\} = 0 \tag{7.5}$$

The subspace with DOFs eliminated is spanned by

$$[T]_{N\times(N-N_e)} = \left[\begin{array}{c} I_{(N-N_e)\times(N-N_e)} \\ G_{N_e\times(N-N_e)} \end{array}\right] = \left[\begin{array}{c} I \\ -c_e^{-1}c_a \end{array}\right] \tag{7.6}$$

The problem that verifies constraints can also be written using Lagrange multipliers, which leads to

$$\left[\begin{array}{cc} [Z(s)] & \left[\begin{array}{c} -G \\ I \end{array}\right] \\ [-G \; I] & 0 \end{array}\right]\left\{\begin{array}{c} q \\ F_c \end{array}\right\} = \left\{\begin{array}{c} F \\ 0 \end{array}\right\} \tag{7.7}$$

The response can be computed using elimination (equation (7.4)) and the forces needed to verify the constraints (resultant forces) can be assumed to be point forces associated with the eliminated DOF q_e which leads to

$$F_c = [[Z_{ea}(s)] + Z_{ee}(s)\,[G]]\,\{q\} - F_e = \left[T_e^T Z(s) T\right]\{q_a\} - T_e^T F \tag{7.8}$$

A common approximation is to ignore viscous and inertia terms in the resultant, that is assume $T_e^T Z(s) T \approx T_e^T K T$.

7.14.2 Local coordinates

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

$$\{q_{all,global}\} = [c_{GL}]\,\{q_{all,local}\} \tag{7.9}$$

Constraints (`mpc, rigid, ...`) are defined in local coordinates, that is they correspond to

$$\{q_{all,local}\} = [T_{local}]\,\{q_{master,local}\} \tag{7.10}$$

with $q_{master,local}$ master DOFs (DOFs in `Case.DOF`) defined in the local coordinate system and the `Case.T` corresponding to

$$\{q_{all,global}\} = [T]\{q_{master,local}\} = [c_{GL}][T_{local}]\{q_{master,local}\} \tag{7.11}$$

As a result, model matrices before constraint elimination (with `NoT`) are expected to be defined in the global response system, while the projected matrix $T^T MT$ are defined in local coordinates.

`celas` use local coordinate information for their definition. `cbush` are defined in global coordinates but allow definition of orientation through the element `CID`.

An example of rigid links in local coordinates can be found in `se_gimbal('ScriptCgl')`.

7.14.3 Enforced displacement

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

7.14.4 Resolution as MPC and penalization transformation

Whatever the constraint formulation it requires a transformation into an explicit multiple point constraint during the resolution. This transformation is accessible for `RBE3` and `rigid`constraints, a cleaned resolution of `MPC` constraints is also accessible using `fe_mpc`.

- `RBE3c` provides the resolution for `RBE3` constraints.
- `RigidC` provides the resolution for `rigid`constraints.
- `MPCc` provides the resolution for `MPC` constraints.

The output is of the format struct with fields

- `c` the constraint matrix.
- `DOF` the DOF vector relative to the constraint.
- `slave` slave DOF indices in `DOF`.

Such format allows the user to transform a constraint into a penalization using the constraint matrix as an observation matrix. One can indeed introduce for each constraint equation a force penalizing its violation through a coefficient `kc` so that $\{f\}_{penal} = kc\,[c]_{N_c\times N}\{q\}_{N\times 1}$. This can be written by means of a symmetric stiffness matrix $[k_{penal}]_{N\times N} = kc\,[c]^T[\mathcal{I}]_{N_c\times N_c}[c]_{N_c\times N}$ added to the system stiffness.

```
% Transformation of a constraint into a penalty
% Generation of a screw model example
model=demosdt('demoscrew layer 1 40 20 3 3 space .2 layer 2 40 20 4');
% Model a screw connection with a RBE3 constraint
% see sdtweb fe_case.html#ConnectionScrew
r1=struct('Origin',[20 10 0],'axis',[0 0 1],'radius',3, ...
 'planes',[0 0 111 1 0;3 0 111 1 0;   % [z0 type ProId zTol rTol]
           5.2 0 112 1 6; 7.2 0 112 1 6], ...
 'MatProId',[101 101],'rigid',[Inf abs('rigid')],'NewNode',0);
r1.planes(:,2)=1; % RBE3
mo2=fe_caseg('ConnectionScrew',model,'screw1',r1);
% display the connection in feplot
cf=feplot(mo2);fecom('colordatamat -alpha .1');

% Replace RBE3 by a penalized coupling
% Get the constraint matrix
r1=fe_mpc('rbe3c',mo2,'screw1');
% remove the RBE3 constraint
mo2=fe_case(mo2,'reset');
% Generate the penalization stiffness with default kc
kc=sdtdef('kcelas');
SE=struct('DOF',r1.DOF,'Opt',[1;1],...
 'K',{{feutilb('tkt',r1.c,kc*speye(length(r1.slave)))}});
% Instance the superelement in the model
mo2=fesuper('seadd -unique 1 1 screw1',mo2,SE,[1 1]);

% Compute the system modes
def=fe_eig(cf.mdl,[5 20 1e3]);
```

7.14.5 Low level examples

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

```
% Low level handling of constraints
 femesh('reset'); model = femesh('test 2bay');
 [m,k,mdof]=fe_mknl(model)
```

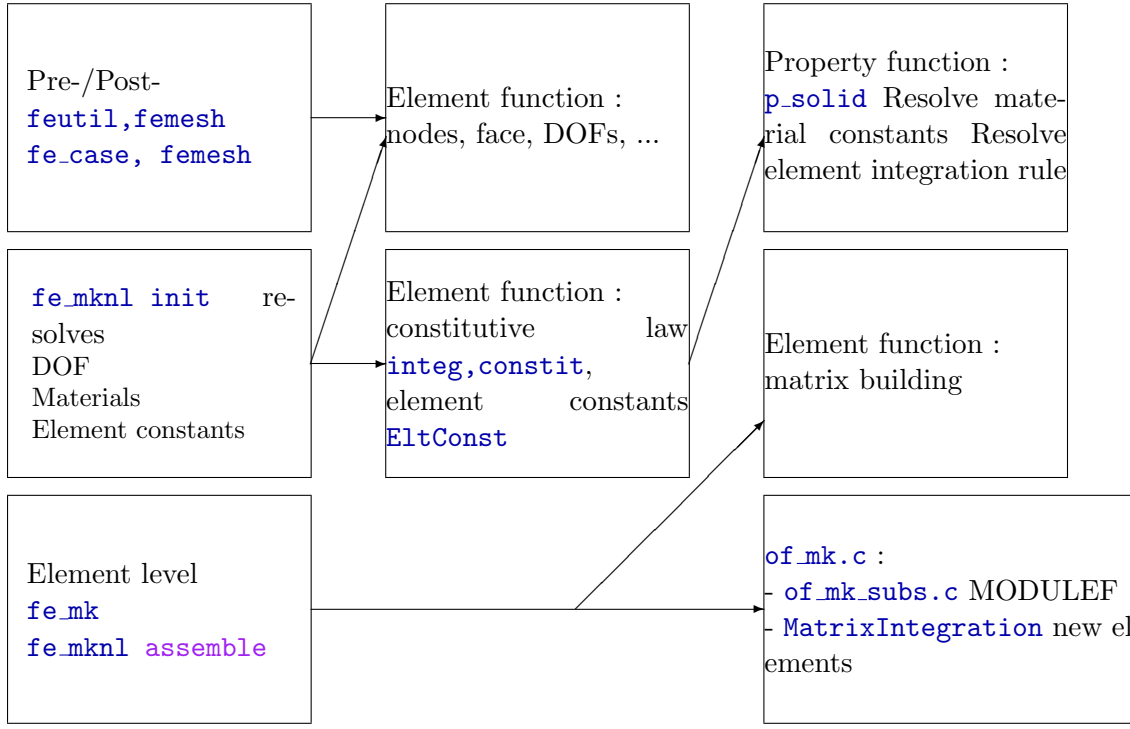
```
i1 = femesh('findnode x==0');
adof1 = fe_c(mdof,i1,'dof',1);              % clamp edge
adof2 = fe_c(mdof,[.03 .04 .05]','dof',1); % 2-D motion
adof = fe_c(mdof,[adof1;adof2],'dof',2);

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

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

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

% select the rigid edge and set its properties
femesh(';selelt group1 & seledge & innode {x==0};addsel');
femesh('setgroup2 name rigid');
FEelt(femesh('findelt group2'),3)=123456;
FEelt(femesh('findelt group2'),4)=0;
model=femesh;

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

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

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

7.15 Internal data structure reference

7.15.1 Element functions and C functionality



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

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

For example, in the assembly step, `fe_mk` analyzes all the groups of elements. For each group, `fe_mk` gets its element type (*bar1*, *hexa8*, ...) and then calls the associated element function.
First of all, `fe_mk` calls the element function to know what is the right call form to compute the elementary matrices (`eCall=elem0('matcall')` or `eCall=elem0('call')`, see section 7.16.6 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.

7.15.2 Standard names in assembly routines

cEGI	vector of element property row indices of the current element group (without the group header)
constit	real (double) valued constitutive information. The constit for each group is stored in Case.GroupInfo{jGroup,4};.
def.def	vector of deformation at DOFs. This is used for non-linear, stress or energy computation calls that need displacement information.
EGID	Element Group Identifier of the current element group (different from jGroup if an EGID is declared).
elt	model description matrix. The element property row of the current element is given by elt(cEGI(jElt),:) which should appear in the calling format eCall of your element function.
ElemF	name of element function or name of superelement
ElemP	parent name (used by femesh in particular to allow property inheritance)
gstate	real (double) valued element state information.
integ	int32 valued constitutive information.
jElt	number of the current element in cEGI
jGroup	number of the current element group (order in the element matrix). [EGroup,nGroup]=getegroup(elt); finds the number of groups and group start indices.
nodeE	nodes of the current element. In the compiled functions, NodeId is stored in column 4, followed by the values at each node given in the InfoAtNode. The position of known columns is identified by the InfoAtNode.lab labels (the associated integer code is found with comstr('lab',-32)). Of particular interest are • v1x (first vector of material orientation, which is assumed to be followed by v1y,v1z and for 3D orientation v2x,y,z), see stack entry info,EltOrient • v3x,v3y,v3z for normal maps • T is used for temperature (stack entry info,RefTemp)
NNode	node identification reindexing vector. NNode(ID) gives the row index (in the node matrix) of the nodes with identification numbers ID. You may use this to extract nodes in the node matrix using something like node(NNode(elt(cEGI(jElt),[1 2])),:) which will extract the two nodes with numbers given in columns 1 and 2 of the current element row (an error occurs if one of those nodes is not in node). This can be built using NNode=sparse(node(:,1),1,1:size(node,1).
pointers	one column per element in the current group gives.

7.15.3 Case.GroupInfo cell array

The meaning of the columns of GroupInfo is as follows

DofPos Pointers Integ Constit gstate ElMap InfoAtNode EltConst

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

pointers int32 matrix whose columns describe information each element of the group. Pointers has one column per element giving
[OutSize1 OutSize2 u3 NdNRule MatDes IntegOffset ConstitOffset StateOffset u9 u10]
Outsize1 size of element matrix (for elements issued from MODULEF), zero otherwise.
MatDes type of desired output. See the MatType section for a current list.
IntegOffset gives the starting index (first element is 0) of integer options for the current element in integ.
ConstitOffset gives the starting index (first element is 0) of real options for the current element in constit.

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

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

`gstate` a curve with field `.Y` describing the internal state of each element in the group. Typical dimensions stress, integration points, elements so that `.Y` has size $Nstrain \times Nw \times NElt$. The labels in `.X{1}` can be used to find positions in the `.Y` matrix. The `.X{2}` should contain the gauss point locations within the reference element. Automated generation of initial states is discussed in section 7.13 .
Users are of course free to add any appropriate value for their own elements, a typical application is the storage of internal variables. For an example of `gstate` initialization see `fe_stress` thermal.
the old format with a `double` matrix with one column per element is still supported but will be phased out.

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

`InfoAtNode` a structure with `.NodePos (int32)` with as many columns as elements in the group giving column positions in a `.data` field. Each row in `.data` corresponds to a field that should be described by a cell array of string in `.lab` used to identify fields in assembly, see `nodeE`. Initialization for a given element type is done the `GroupInit` phase, which uses `pro.MAP` fields (see section 7.13). Typical labels for orientation are `{'v1x','v1y','v1z','v2x','v2y','v2z'}`
Obsolete format : `double` matrix whose **rows** describe information at element nodes (as many columns as nodes in the model).

`EltConst` `struct` used to store element formulation information (integration rule, constitutive matrix topology, etc.) Details on this data structure are given in section 7.15.4 .

7.15.4 Element constants data structure

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

`.N`	$nw \times Nnode$ shape functions at integration points
`.Nr`	$nw \times Nnode$ derivative of shape function with respect to the first reference coordinate r
`.Ns`	$nw \times Nnode$ derivative of shape function with respect to the second reference coordinate s
`.Nt`	$nw \times Nnode$ derivative of shape function with respect to the second reference coordinate t
`.NDN`	$Nshape \times nw(1 + Ndim)$ memory allocation to store the shape functions and their derivatives with respect to physical coordinates $[N\ N,x\ N,y\ N,z]$. `of_mk` currently supports the following geometry rules `3` 3D volume, `2` 2D volume, `23` 3D surface, `13` 3D line (see `integrules BuildNDN` for calling formats). Cylindrical and spherical coordinates are not currently supported. In the case of rule `31` (hyperelastic elements), the storage scheme is modified to be $(1+Ndim) \times Nshape \times nw$ which preserves data locality better.
`.jdet`	Nw memory allocation to store the determinant of the jacobian matrix at integration points.
`.bas`	$9 \times Nw$ memory allocation to store local material basis. This is in particular used for 3D surface rules where components `6:9` of each column give the normal.
`.Nw`	number of integration points for output (inferior to `size(EltConst.N,1)` when different rules are used inside a single element)
`.Nnode`	number of nodes (equal to `size(EltConst.N,2)=size(EltConst.NDN,1)`)
`.xi`	$Nnode \times 3$ reference vertex coordinates
`.VectMap`	index vector giving DOF positions in external sort. This is needed for RHS computations.
`.CTable`	low level interpolation of constitutive relation based on field values. Storage as a double vector is given by `[Ntables CurrentValues (Ntables x 7) tables]` with `CurrentValues` giving `[i1 xi si xstartpos Nx nodeEfield constit(pos_Matlab)]`. Implementation is provided for `m_elastic` to account for temperature dependence, `fe_mat` to generate interpolated properties.

7.16 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.

7.16.1 Generic compiled linear and non-linear elements

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

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

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

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

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

7.16.2 What is done in the element function

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

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

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

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

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

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

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

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

  For elements that have an internal orientation (shells, beams, etc.) it is expected that orientation maps are built during this command (see beam1t, ...). Note, that the 'info','EltOrient' stack entry can also be used for that purpose.

- standard topology information (commands node, dof, prop, line, patch, face, edge, parent) see section 7.16.6 .

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

7.16.3 What is done in the property function

p_fcn

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

BuidConstit

As shown in section 7.15.1 and detailed under fe_mknl the FEM initialization phase needs to resolve

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

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

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

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

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

Const

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

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

***_fcn**

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

Subtype

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

database

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

dbval

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

PropertyUnitType

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

This mechanism is used to automate unit conversions in `fe_mat` Convert.

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

Cell arrays describing each subtype give

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

7.16.4 Compiled element families in of_mk

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

Generic multi-physic linear elements

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

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

Elements of this family are standard element functions (see section 7.16) and the element functions must thus return `node`, `prop`, `dof`, `line`, `patch`, `edge`, `face`, and `parent` values. The specificity is that all information needed to integrate the element is stored in an `EltConst` data structure that is initialized during the `fe_mknl GroupInit` phase.

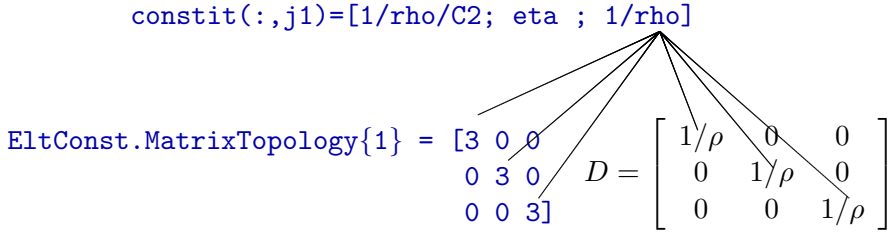
For DOF definitions, the family uses an internal DOF sort where each field is given at all nodes sequentially $1x2x...8x1y...8y...$ while the more classical sort by node $1x1y...2x...$ is still used for external access (internal and external DOF sorting are discussed in section 7.16.6).

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

$$k^{(e)} = \sum_{ji,jj} \sum_{jw} \left[\{B_{ji}\} D_{ji\ jk}(w(jw)) \{B_{jj}\}^T \right] J(w(jw)) W((jw)) \tag{7.12}$$

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

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



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

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

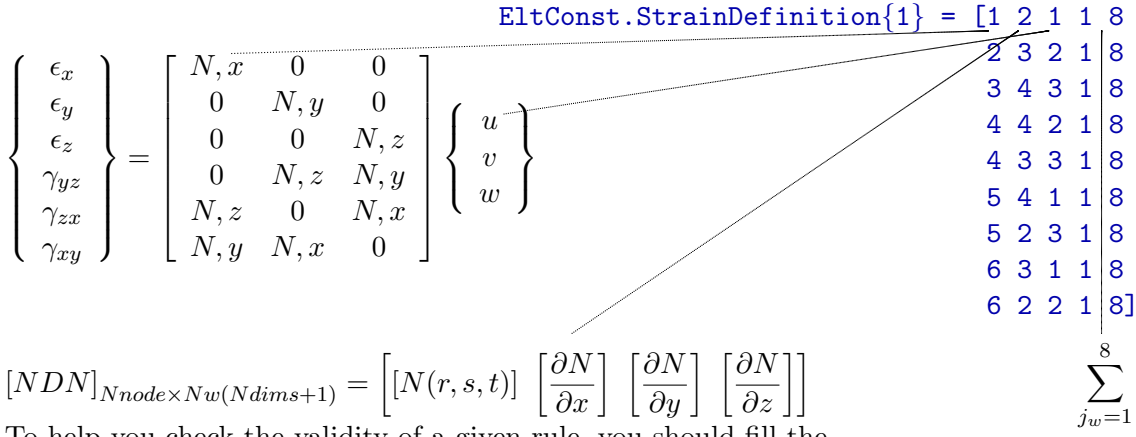
$$[NDN]_{Nnode \times Nw(Ndims+1)} = \left[[N(r,s,t)] \left[\frac{\partial N}{\partial x}\right] \left[\frac{\partial N}{\partial y}\right] \left[\frac{\partial N}{\partial z}\right] \right] \tag{7.13}$$

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

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

`EltConst.StrainDefinition{jType}` with each row describing `row, NDNBloc, DOF, NwStart, NwTot` giving the strain component number (these can be repeated since a given strain component can combine more than one field), the block column in NDN (block 1 is N, 4 is $\partial N/\partial z$, a nega-

tive number can be used to specify $-N$, ...), the field number, and the starting integration point associated with this strain component and the number of integration points needed to assemble the matrix. The default for `NwStart NwTot` is `1, Nw` but this formalism allows for differentiation of the integration strategies for various fields. The figure below illustrates this construction for classical mechanical strains.



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

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

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

Phases in of_mk.c matrix integration

The core of element computations is the `matrixintegration` command that computes and assembles a group of elements.

After a number of inits, one enters the loop over elements.

The `nodeE` matrix, containing *field at element nodes*, is filled with information at the element nodes as columns. The first 3 columns are positions. Column 4 is reserved for node numbers in case a callback to MATLAB makes use of the information. The following columns are based on the `InfoAtNode` structure whos indexing strategy is compatible with both continuous and discontinuous fields at each node. See `sdtweb elem0('get_nodeE')` for details.

Initialization of InfoAtNode is performed with fe_mknl('Init -gstate') calls. The m_elasticAtNodeGState command is an illustration of init used to interpolate material properties in volume elements.

The defe vector/matrix contains the values at the current element DOF of the provided deformation(s).

Generic RHS computations

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

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

Two main strategies are supported where the fields needed for the integration of loads are stored either as columns of dc.def (for fields that can defined on DOFs of the model) or as nodeE columns.

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

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

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

- volume forces not proportional to density

$$\int_{\Omega_0} f_v(x).du(x) = \{F_v\}_k = \sum_{j_w} \left(\{N_k(j_w)\}\{N_j(j_w)\} f_v(x_j)\right) J(j_w) W(j_w) \tag{7.14}$$

  are thus described by

```
opt.RhsDefinition=int32( ...
```

```
[101 0 3 0     0 0 -1    rule+[-1 0];
 101 1 3 0     1 0 -1    rule+[-1 0];
 101 2 3 0     2 0 -1    rule+[-1 0]]);
```

for 3D solids (see p_solid).

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

$$\begin{array}{rcl} F_m & = & \int_{\partial\Omega_0} p(x) n_m(x).dv(x) \\ & = & \sum_{jw} \left(\{N_k(j_w)\}\{N_j(j_w)\} p(x_j) n_m\right) J(j_w) W(j_w) \end{array} \tag{7.15}$$

- inertia forces (volume forces proportional to density)

$$F = \int_{\Omega_0} \rho(x) f_v(x).dv(x) \tag{7.16}$$

- stress forces (will be documented later)

Large transformation linear elasticity

Elastic3DNL fully anisotropic elastic elements in geometrically non-linear mechanics problems. Element matrix are implemented in the of_mk.c MatrixIntegration command with StrategyType=2 for the linear tangent matrix (MatType=5). Other computations are performed using generic elements (section 7.16.4) (mass MatType=2). This formulation family has been tested for the prediction of vibration responses under static pre-load.

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

Hyperelasticity

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

$$[NDN]_{(Ndims+1)\times Nnode(Nw)} = \left[[NDN]^{jw}\right] \tag{7.17}$$

with

$$[NDN]^{jw}_{(Ndims+1)\times Nnode} = \begin{bmatrix} [N(r,s,t)]_{jw} \\ \left[\frac{\partial N}{\partial x}\right]_{jw} \\ \left[\frac{\partial N}{\partial y}\right]_{jw} \\ \left[\frac{\partial N}{\partial z}\right]_{jw} \end{bmatrix} \tag{7.18}$$

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

Other computations are performed using generic elements (section 7.16.4) (mass `MatType=2`). This formulation family has been tested for the `RivlinCube` test.

Stress post-processing is not yet implemented for hyperelastic media.

7.16.5 Non-linear iterations, what is done in `of_mk`

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

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

At a given iteration, one resets the RHS and performs a single `fe_mknl` call that returns the current non-linear matrix and replaces the RHS by its current value (note that `fe_mknl` actually modifies the input argument `dc` which is not an normal MATLAB behavior but is needed here for performance)

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

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

- `Linear` multiphysics elements of arbitrary forms, see section 7.16.4

- `Elastic3DNL` general elastic elements for large transformation, see section 7.16.4
- `Hyperelastic` elements for large transformation problems. see section 7.16.4 . These elements have been tested through the `RivlinCube` example.

7.16.6 Element function command reference

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

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

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

General element information

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

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

For element functions the nominal format is

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

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

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

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

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

The expected outputs to these calls are detailed below.

`call,matcall`

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

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

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

```
[k1,m1]=elem0(nodeE,elt(cEGI(jElt),:),...
              pointers(:,jElt),integ,constit,elmap);
```

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

k1	element matrix (must always be returned, for opt(1)==0 it should be the stiffness, otherwise it is expected to be the type of matrix given by opt(1))
m1	element mass matrix (optional, returned for opt(1)==0, see below)

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

fe_mk and fe_mknl variables that can be used as *input arguments to element function* are listed in section 7.15.2 .

dof, dofcall

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

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

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

Elements may use different DOF sorting for their internal computations.

edge,face,patch,line,sci_face

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

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

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

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

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

`sci_face` is the equivalent of `patch` for use in the SCILAB implementation of *OpenFEM*. The difference between `patch` and `sci_face` is that, in SCILAB, a face must be described with 3 or 4 nodes. That means that, for a two nodes element, the last node must be repeated (in generality, `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 :

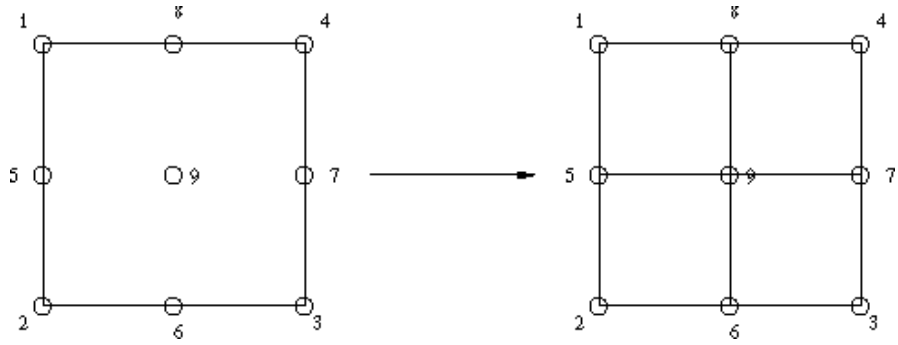


Figure 7.1: Lower order patch representation of a 9 node quadrilateral

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

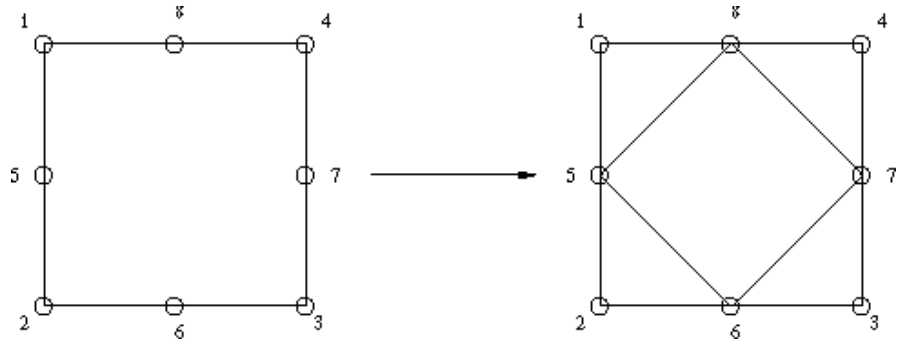


Figure 7.2: Lower order patch representation of a 8 node quadrilateral

integinfo, BuildConstit

`integinfo`, `BuildConstit` are commands to resolve constants in elements and `p_function` respectively.

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

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

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

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

`node`

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

`prop`

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

parent

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

rhscall

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

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

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

Matrix, load and stress computations

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

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

A typical element assembly section is as follows :

```
% elem0 matrix assembly section

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

```
% outputs are [k,m] for opt(1)==0
%             [mat] for other opt(1)
switch point(5)
case 0
 [out,out1] = ... % place stiffness in out and mass in out1
case 1
  out= ...  % compute stiffness
case 2
  out= ...  % compute mass
case 100
  out= ...  % compute right hand side
case 200
  out= ...  % compute stress  ...
otherwise
  error('Not a supported matrix type');
end
```

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

There is currently no automated mechanism to allow users to integrate such computations for their own elements without modifying fe_load and fe_stress, 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_stress will then evaluate these calls for each element.

7.17 Variable names and programming rules (syntax)

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

All SDT functions are segmented and tagged so that the function structure is clearly identified. Its tree structure can be displayed and browsable through the sdtweb _taglist interface. You should produce code compatible with this browser including tags (string beginning by # in a comment), in particular at each command of your function.

In addition, input parsing section 7.17.3 and some utilities for directory handling section 7.17.4 , post-treatment display section 7.17.4 and figure formatting/capturing section 7.17.4 have been standardized.

7.17.1 Variable naming conventions

Standardized variable names are

`carg`	index of current argument. For functions with variable number of inputs, one seeks the next argument with `NewArg=varargin{carg};carg=carg+1;`
`CAM`, `Cam`	string command to be interpreted. `Cam` is the lower case version of `CAM`.
`j1,j2,j3` ...	loop indices.
`jGroup,jElt,jW`	indices for element groups, elements, integration points. For code samples use `help('getegroup')`
`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

`model, mo1, mo2`	SDT model structures.
...	
`node,FEnode, n1,`	nodes, FEnode is reserved as a global variable.
`n2` ...	
`elt, FEelt, el1,`	elements, FEelt is reserved as a global variable.
`el2` ...	
`EGroup, nGroup`	starting index of each group and number of groups in an element structure, see `help('getegroup')`.
`cEGI`	index of elements for a given group in an element structure, see `help('getegroup')`.
`NNode`	reindexing vector, verifies `NodeInd=NNode(NodeId)`. Can be built using `NNode=sparse(node(:,1),1,1:size(node,1))`.
`nd`	reindexing object for DOF, verifies `DofPos=feval(nd.getPosFcn,nd,DOF)`. Is built using `nd=feval(fe_mknl('@getPosFromNd'),[],DOF);`
`RunOpt`	run options, a structure used to store options that are used in a command. `RO` can also be used.
`adof`	current active DOF vector.
`cf`	pointer to a `feplot` figure.
`gf, uf, ga, ua,`	respectively handle and userdata to a figure, handle and userdata to an axis,
`go, uo`	handle and userdata to a graphics subobject.
`gc, evt`	respectively active object and associated event in Java triggered callbacks.

7.17.2 Coding style

The coding styles convention are detailed in the example below.

- Tags for taglist are marked with the `#` token, not to interfere with `pragma` tokens, ensure that it is not directly following a `%`, but leave at least one space.
  - The tag level can be specified by placing `-i` at the end of the line, `i` being the level. If not each tag is assumed to be level 1. Tags with lines finishing by `- - -` or after the `#SubFunc` tag are assumed level 2.
  - By default, the taglist will concatenate consecutive tags with the same starting letters, the subsequent tags will thus be shifted.
- Code sections are usually delimited using the cell display `%%`.
- The first input argument should be a string whose parsing will determine the command to execute and associated command options.

- An error should be returned if the command is unknown.
- Access from the outside to subfunction handles should be made possible through a call `suf=my_func('@my_sub_fun')`.
- Subversion tags should be present to allow easy administration using `cvs` or `svn`, in a unique command `cvs`, that will output a string containing the `cvs` or `svn` tags.

```
function [out,out1,out2,out3]=my_func(varargin);

% Here you should place your help
% SDT functions always use varargin for input and [out,out1, ...] for
% output.

% ask MATLAB to avoid some warnings the in the editor MLint
%#ok<*NASGU,*ASGLU,*CTCH,*TRYNC,*NOSEM>

% Get the command in varargin{1} and strip front/end blanks with comstr
% CAM is as input, Cam is lower case.
[CAM,Cam]=comstr(varargin{1},1);carg=2;


%% #Top : main level command Top  ------------------------------
%  the %% is to use Matlab cell, while #Top is a sdtweb _taglist tag
%  by default tags are set to level 1
% Now test that Cam starts by 'top' and then strip 3 characters and trim (+1)
if comstr(Cam,'top');[CAM,Cam]=comstr(CAM,4);

 if comstr(Cam,'manual')
 %% #TopLevel2 : subcommand level 2 - - - - - - - - - -2
 % - - - tells sdtweb this is a level 2 tag
 % if sufficies to end the line with -2 in practice
 % any other level can be used by adding a higher number at the end of the tag line

  % recover other inputs
  r1=varargin{carg}; carg=carg+1; % get input and increment counter
  % get optionnal input arguments
  if carg<=nargin; r2=carargin{carg}; carg=carg+1; else; r2=[]; end
  % ...
```

```
 %% #TopEnd -2
 else; error('Top%s unknown',CAM);
 end
%% #End : typical commands placed at end of function
elseif comstr(Cam,'@');out=eval(CAM);
elseif comstr(Cam,'cvs')
 out='$Revision: 1.17 $  $Date: 2019/02/15 17:37:30 $';
else; error('my_func %s unknown',CAM);
end

%% #SubFunc : indicates start of subfunctions to taglist parsing
%% #my_sub_fun - - ----------------------------------------
function out=my_sub_fun(varargin)
```

7.17.3 Input parsing conventions

Passing command options is a critical feature to enable little behavior alteration as function of the user needs although most of the functionality is the same. This allows in particular limiting code duplication.

From the input `CAM` variable, command option parsing utilities have been defined and standardized. The goal is to build a run option structure from the input command string while keeping the possibility to provide it as an extra argument.

The command parsing code is then

```
% Usual run options handling
% first possible to recover in extra input
if carg>nargin||~isstruct(varargin{carg});RO=struct;
else;RO=varargin{carg};carg=carg+1;
end
% then parse CAM for command options,
%  and assign default values to unspecified options
%  values declared prior to the paramedit call are not overriden
[RO,st,CAM]=cingui('paramedit -DoClean',[ ...
 'param(val#%g#"Description")' ...
 'token(#3#"token modes does...")' ...
 '-parS("string"#%s#"parS modes available...")' ...
```

```
],{RO,CAM}); Cam=lower(CAM);
```

The paramEdit call from cingui performs standard operations for each token in the second input string of the command. Each token follows the format token(val#typ#"info"), and will generate a case sensitive field token in the structure RO. val is a default value that is applied if the field token is missing before the call. info is a string providing information on the token effect. typ tells the type of input that should be parsed after the token, with the following rules:

- 3 Only checks for the presence of token in the command without any other value. Sets field token to 1(double) if found, 0(as double) if not. val must remain empty. *e.g.* Top token, will set RO.token=1.
- 31 Behaves as type 3 but also checks for an optional integer input. Sets field token to 1(double) if found, 0(as double) if not, or to the found integer if found. val must remain empty. *e.g.* Top token 2 will set RO.token=2, and Top token will set RO.token=1.
- %g Checks for token followed by a float. If found RO.token is set to the float, if no float is found the field is left empty. If the token is not found, the default value val is set. *e.g.* Top token 3.14 will set RO.token=3.14.
- %i Checks for token followed by an integer. If found RO.token is set to the integer, if no integer is found the field is left empty. If the token is not found, the default value val is set. *e.g.* Top token 31 will set RO.token=31.
- %s Checks for token followed by a string (delimited by two "). If found RO.token is set to the string, if no string is found the field is left empty. If the token is not found, the default value val is set. *e.g.* Top token"test" will set RO.token='test'. Note that for this type if val is not empty one defines the token as token("val"#%s#"info"), but if val is empty, one should use token(#%s#"info").

The output CAM has been stripped from any parsed data.

The format -token(val#typ#"info") will require the presence of -token in the command to generate the token field in RO.

By convention, to handle interferences between the extra input argument RO and default values overriding, any field present in RO prior to calling paramEdit will be left unchanged by the command.

7.17.4 Commands associated to project application functions

The development of project application functions follow some must have such as project directory handling section 7.17.4 , post-treatment handling section 7.17.4 , image capture generation section 7.17.4 . Some of these steps have been standardized over the years, which are documented in the present sections.

wd,fname

The files relative to a specific application are usually stored in a specific file arborescence. It is thus useful to access standardly defined save directories in a robust manner, regardless of the operating system or the user. Standard applications developed by SDTools usually involve a user defined root directory from which the following subdirectories are defined

- `m` contains the project source code.
- `tex` contains the project documentation source code.
- `mat` contains reference data files.
- `plots` contains the image captures.
- `doc` contains other project support documentation.

Each of these directories may contain any further arborescence to class data as desired.

To allow efficient recovery of a specific subdirectory or file in the final project file architecture, `sdtweb` provides commands in its utilities (see `sdtweb Utils`) that should be used by the main project function to search the project architecture subdirectories.

The `wd` command should package a search in its known subdirectories.

```
%% #wd ------------------------------------------------
elseif comstr(Cam,'wd')

if nargin==1 % output the possible root directories
 % assume this function is stored in root/m
 out=fileparts(which('my_func'));
 % possibly add specific root dirs outside the project
 % should be better handled with a preference
 wd2={'/p/my_files'}; % add as many as needed
 out=[out wd2];

else % get the subdirectory searched
```

```
 wd1=varargin{carg}; carg=carg+1;
 % get the project root directory (several ones admitted)
 wd0=my_func('wd');
 % find the subdirectory
 out=sdtweb('_wd',wd0,wd1);
end
```

The fname command should package a file search in the known subdirectories

```
%% #fname -----------------------------------------------
elseif comstr(Cam,'fname')
fname=varargin{carg}; carg=carg+1;
% get the available root directories
wd=my_func('wd');
% search for the file
out=sdtweb('_fname',fname,wd);
```

view

The generation of displayed post-treatments should be handled by a command named View, that will centralize the feplot manipulations required to generate *ad hoc* displays. Variations of display are handled in the command, first and second input should be the feplot pointer and optionally a deformation data.

- Handling of legend (location, labels, ...) can be performed by defining a Legend field to deformation curves, see comgui def.Legend for more details.
- Handling of colorbars and their legends can be performed using fecom ColorBar and fecom ColorLegend commands.
- Stress post-treatments can be handled through a fe_caseg StressCut command.
- Energy post-treatment can be handled through fe_stress Ener and their corresponding display through fe_stress feplot
- Handling of color scales can be handled with fecom ColorScale.

A sample call to be handled by the view command could then be.

```
my_project('ViewUpStress',cf);
```

im

The generation of image captures from figures (feplot iiplot or standard MATLAB figures) should be handled by a command named im, that will centralize formatting and saving. This command should

- Provide figure formatting data for implemented modes
- Perform figure formatting according to a required mode
- Perform figure capture and save to an appropriate directory

For details on figure formatting, see comgui objSet, for details on figure naming strategy see comgui ImFtitle, for low level image capturing calls, see comgui ImWrite.

A suggested layout for the im command of a sample my_func function is then

```
%% #im : figure formatting ------------------------------------------------
elseif comstr(Cam,'im')
% sdt_table_generation('Rep{SmallWide}');comstr(ans,-30)

 if nargin==2 % generate the calling string
   pw0=pwd;
   if isfield(varargin{2},'ch') % multiple generation with imwrite ch
    RO=varargin{2};cf=feplot;
    % Create an possibly change to directory
    sdtkey('mkdircd',my_func('wd','plots',sscanf(cf.mdl.name,'%s',1)));
    RO.RelPath=1; % Save links with path relative to current position
    RO=iicom(cf,'imwrite',RO);
    fid=fopen('index.html','w');fprintf(fid,'%s',RO.Out{:});fclose(fid);
    cd(pw0);

   elseif ~ischar(varargin{2}); % Apply reshaping to figure
     gf=varargin{2};if ~ishandle(gf);figure(gf);plot([0 1]);end
     cingui('objset',gf,my_func(CAM))
     % if feplot, center the display
     if strcmpi(get(gf,'tag'),'feplot');iimouse('resetvie');end

   elseif strcmpi(varargin{2},'.') % if '.' get automatic naming
    st=sprintf('imwrite-objSet"@my_func(''%s'')"-ftitle',varargin{1});
    comgui(st);
```

```
   else
    cd(my_func('wd','plots'));
    st=sprintf('imwrite-objSet"@my_func(''%s'')"-ftitle%s',varargin{1:2});
    comgui(st);
    cd(pw0);
   end

 elseif comstr(Cam,'imw1') % Figure formatting options for w1
    out={'position',[NaN,NaN,450*[1.5 2]],'paperpositionmode','auto', ...
        '@exclude',{'legend.*'},'@text',{'FontSize',14}, ...
        '@axes',{'FontSize',14,'box','on'}, ...
        '@ylabel',{'FontSize',14,'units','normalized'}, ...
        '@zlabel',{'FontSize',14,'units','normalized'}, ...
        '@title',{'FontSize',14}, ...
        '@line',{'linewidth',1}, ...
        '@xlabel',{'FontSize',14,'units','normalized'}};

% elseif ... use as many commands as needed

 else; error('%s unknown',CAM);
 end
```

This way, the following tasks can be easily performed

```
% Im calls for figure capturing
gf=figure(1); plot([1 0]);
% Capture an image from figure 1 with formatting w1 and named test.png
my_func('imw1','test.png');
% Capture an image from figure 1 with formatting w1 with an automatic name
my_func('imw1','.');
% Format figure 1 according to w1 options
my_func('imw1',gf);
% Get formatting options for w1
r1=my_func('imw1');
```

7.17.5 Commands associated to tutorials

In a `training` function or in any function where a tutorial could be executed, the syntax is the following

```
 elseif comstr(Cam,'tuto')
  %% #Tuto (implement standard behaviour of tuto command) -1
   % Execute the tutorial with CAM commands or open the tuto tree if empty CAM
  eval(sdtweb('_tuto',struct('file','current_function_name','CAM',CAM)));
  if nargout==0; clear out; end
 elseif comstr(Cam,'tutoname')
  %% #TutoTutoname-2
  % See sdtweb('LinkToHTML') % Open the HTML corresponding to the tutorial

  %% Step 1 : Description of step1
  % See sdtweb('LinkToHTML') % Open HTML detailed doc related to this step

  %% Step 1.1 : Description of substep 1.1

  % Code to execute correponding to Step 1.1

  %% Step 1.2 : Description of substep 1.1

  % Code to execute correponding to Step 1.2

  % Step 2 : Description of step2
  % See sdtweb('LinkToHTML') % Open HTML detailed doc related to this step

  % Code to execute correponding to Step 2

  %% EndTuto

 elseif comstr(Cam,'tutoname2')
  %% #TutoTutoname2-2
  % See sdtweb('LinkToHTML') % Open the HTML corresponding to the tutorial

  %% EndTuto

% elseif ... use as many commands as needed
```

This way, the following commands are usually executed :

```
% Open the tree containing all the tutorial and clickable buttons
my_func('Tuto');
% Execute the whole tutorial (useful for test auto)
```

```
my_func('TutoTutoname');
% Execute a tutorial up to a given step (here section 2.3)
my_func('TutoTutoname -s2.3');
```

7.18 Criteria with CritFcn

SDT supports the use of various criteria to be applied on data. The default `CritFcn` implementation is present in `fegui`. The fields of a `CritFcn` structure are

- `.cmap` colormap.
- `.clevel` levels associated with the colors (one more level than the number of colors). If not present, the default is an equal spacing of colors in the [0,1] interval. This field is typically used to color tables.
- `.cback` default color if below the `.clevel` interval. Defaults to white.
- `.llevel` levels associated with line plots.
- `.Fcn` handle to handling function, defaults to `fegui('@CritFcn')`.
- `.imap` *alternative* to `.Fcn` to specify color index by hand.

```
r1=(1:10)'; r1=[r1 sin(r1/max(r1)*pi) cos(r1/max(r1)*pi) ];
% Standard criterion
R1=struct('clevel',linspace(0,1,4),'cmap',eye(3),'Fcn',fegui('@CritFcn'));
% Manual setting of color map
R2=struct('cmap',eye(3),'imap',round((r1(:,3)+1)*3/2));
ua=struct('name','CritFcn','ColumnName',{{'#','val','ind';'','','';
  '0','0.00','.0%'; ...  % Column formatting (java)
  R1,R1,R2}}, ... % Define a CritFcn for coloring
 'setSort',2); % use filter-sort
ua=menu_generation('jpropcontext',ua,'Tab.ExportTable');
%feval(R1.Fcn,'imap',R1,r1)
comstr(r1,-17,'tab',ua)
```

7.19 Legacy information

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

7.19.1 Legacy 2D elements

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

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

Where

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

The xy plane is used with displacement DOFs .01 and .02 given at each node. Element matrix calls are implemented using .c files called by of_mk_subs.c and handled by the element function itself, while load computations are handled by fe_load. For integration rules, see section 7.19.2 . The following elements are supported

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

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

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

7.19.2 Rules for elements in of_mk_subs

hexa8, hexa20

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

The hexa8 element uses the et*3q1d routines.

hexa8 volumes are integrated at 8 Gauss points

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

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

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

hexa8 surfaces are integrated using a 4 point rule

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

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

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

The hexa20 element uses the et*3q2c routines.

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

with

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

with

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

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

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

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

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

penta6, penta15

The penta6 and penta15 elements are the standard 6 node 18 DOF and 15 node 45 DOF pentahedral elements. A derivation of these elements can be found in [43].

The penta6 element uses the et*3r1d routines.

penta6 volumes are integrated at 6 Gauss points

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

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

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

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

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

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

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

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

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

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

Positions and weights of the 21 Gauss point are

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

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

`tetra4, tetra10`

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

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

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

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

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

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

tetra10 volumes are integrated at 15 Gauss points

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

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

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

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

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

tetra10 surfaces are integrated using a 7 point rule

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

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

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

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

q4p (plane stress/strain)

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

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

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

q4p axisymmetric

For surfaces, q4p uses a 4 point rule with

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

For edges, q4p uses a 2 point rule with

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

q5p (plane stress/strain)

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

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

q8p (plane stress/strain)

For surfaces, q8p uses a 9 point rule with

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

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

For edges, q8p uses a 3 point rule with

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

q8p axisymmetric

For surfaces, q8p uses a 9 point rule with

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

For edges, q8p uses a 3 point rule with

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

t3p (plane stress/strain)

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

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

t3p axisymmetric

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

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

t6p (plane stress/strain)

For surfaces, t6p uses a 3 point rule with

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

For edges, t6p uses a 3 point rule

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

t6p axisymmetric

For surfaces, t6p uses a 7 point rule

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

with :

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

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

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

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

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

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

GUI and reporting tools

8.1 Formatting MATLAB graphics and output figures 348
8.1.1 Formatting operations with objSet 349
8.1.2 Persistent data in Project 349
8.1.3 OsDic dictionnary of names styles 350
8.1.4 File name generation with objString 352
8.1.5 Image generation with ImWrite 352
8.2 SDT Tabs 352
8.2.1 Project 353
8.2.2 FEMLink 353
8.2.3 Mode 356
8.2.4 TestBas : position test versus FEM 359
8.2.5 StabD : stabilization diagram 362
8.2.6 Ident : pole tuning 364
8.2.7 MAC : Modal Assurance Criterion display 365
8.3 Non java SDT Tabs (obsolete) 367
8.4 Handling data in the GUI format 367
8.4.1 Parameter/button structure 367
8.4.2 DefBut : parameter/button defaults 369
8.4.3 Reference button file in CSV format 370
8.4.4 Data storage and access 371
8.4.5 Tweaking display 374
8.4.6 Defining an exploration tree 376
8.4.7 Finding CinCell buttons in the GUI with getCell 377
8.5 Interactivity 378
8.5.1 Handling tabs 378
8.5.2 Handling dependencies 378
8.5.3 Dialogs 380

This chapter aims at providing the details and procedures used to build a GUI with SDT. The GUI is based on a formalism where the data and their display is decoupled.

The data considered is a set of parameters preliminary defined through the use of a `csv` file read by `sdt_locale`, quick definitions are supported by `cingui ParamEdit`.

This data is then transformed into a Java object stored as a `v_handle` in the GUI figure. The GUI figure must be named and tagged appropriately to be accessed at any time. Its `Name` and `Tag` are equal and define the figure as unique.

Access to the data parameters is always performed through a `v_handle` call and can be edited using `sdcedit`. Layout of the data can be shaped as desired and displayed under the form of `Tables` in the GUI figure, using `sdt_dialogs` and `cinguj`. The tables are interactive as the user can edit the data parameter fields through the interface. Dependency handling of other parameters as function of the edited one is possible.

8.1 Formatting MATLAB graphics and output figures

SDT implements single `comgui ImWrite` and multiple `iicom ImWrite` image generation mechanisms. The basic process is to

- generate your figure,
- call `comgui objSet` for the initial formatting,
- use `sdtroot Set` to define project information such as the plot output directory.
- use `comgui PlotWd` to predefine output options (directory, file name generation scheme, reformatting for image generation, insertion options for word, ...)

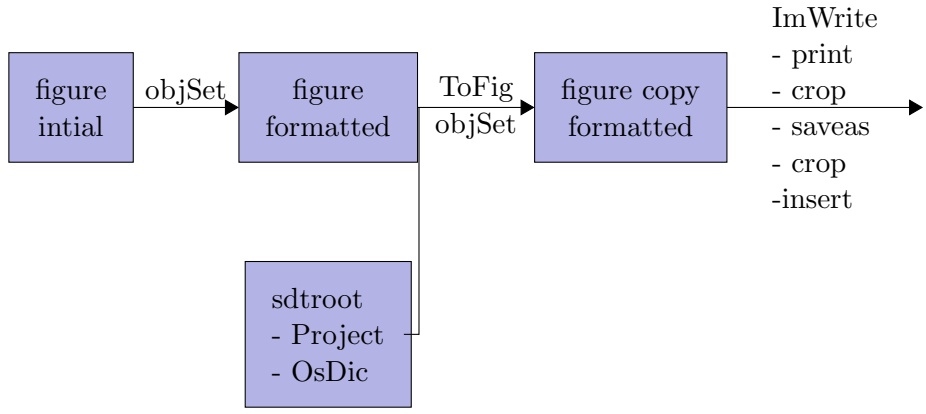


Figure 8.1: Figure generation process

8.1.1 Formatting operations with `objSet`

`cingui('objSet',h,Prop)` groups all formatting operations needed to obtain exactly the figure you want (font size, axes positions, line sequencing, ...) starting from a pointer to a MATLAB graphic `h` and a *style* given as cell array of formatting instructions `Prop`. It is the base SDT mechanism to generalize the MATLAB `set` command.

`Prop` is a cell array of tag-value pairs classical in MATLAB handle properties `comgui objSet` allows three types of modification

- recursion into objects or object search. Thus the property `'@axes'` of a figure is a handle to all axes within this figure or `'@line(2)'` is the second line object.
- expansion is the mechanism where a tag-value pairs is actually replaced by a larger list of tag-value pairs. The definition of styles using `comgui objSet` entries leads to the use of expansion in the form `'@OsDic(SDT Root)',{'val1','val2'}`. This mechanism is key to let the user manage predefined styles.
- Value replacement/verification to enhance basic set commands used by MATLAB. Thus with `'Position',[NaN NaN 500 300]` the lower left corner values shown here as `NaN` are replaced by their current value.

8.1.2 Persistent data in `Project`

The Project tab is initialized using `sdtroot Set` commands. The most commonly used fields are the project and plot directories and file name for export to Word, PowerPoint. Their use is illustrated in the next section.

```
sdtroot('SetProject',struct('ProjectWd',sdtdef('tempdir'), ...
  'root','MyTest'));
```

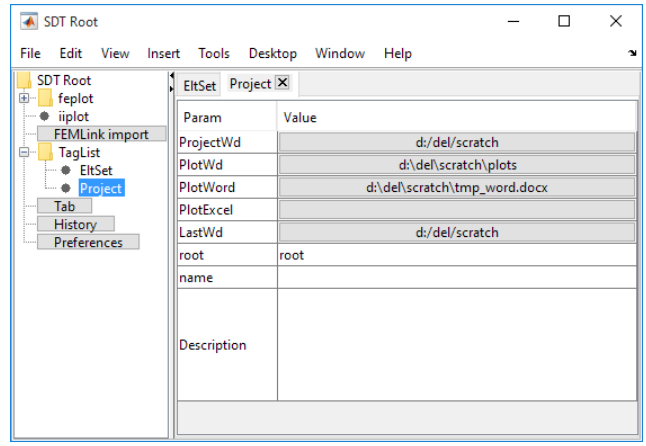


Figure 8.2: Basic project tab

8.1.3 `OsDic` dictionnary of names styles

The `comgui objSet` provides a basic mechanism to provide formatting instructions. As choosing those takes time and for the sake of uniformity it is useful to introduce style sheets, which SDT does using a list of named styles, as shown in figure 8.3.

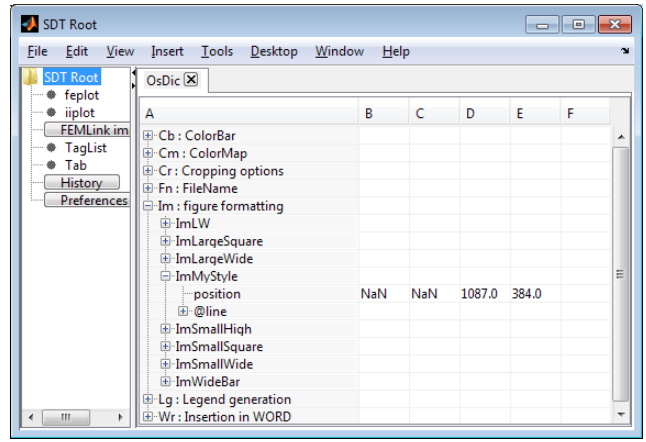


Figure 8.3: Hierarchical view of project styles `sdtroot('InitOsDic')`

Basic implementations of most styles are provided in `d_imw` (see list with `sdtweb('_taglist','d_imw')`). The main categories of styles are

- `Im` : image formatting
  - `SmallWide` for a wide picture (9:16) (landscape style) adapted to reports.
  - `SmallSquare` for a square picture (4:3) adapted to reports.
  - `SmallHigh` for a vertical rectangular picture (9:16) adapted to reports.
  - `LargeWide` for a wide picture (landscape style) adapted to posters.
  - `LargeSquare` for a square picture (4:3) adapted to posters.
  - `WideBar` for a (4:3) landscape style picture. It has the same width than `SmallWide` but is higher, this is mostly convenient for wide `bar` diagrams.
- `Cb` colorbar insertion
- `Cm` colormap definition
- `Cr` image cropping options
- `Fn` file naming strategy. `Fn` + a combination of `R`oot (project root field), `T`(itle) (figure title), `x`label, `y`label, `z`label (figure label), `ii`_legend (see `ii_plp Legend`), `N`ame (cf.data.name), `M`odel (cf.mdl.name)
- `Pr` figure configurations when opening project. See `sdtweb('d_imw','Pr')`

- `Fi feplot` view initialization using a `comgui objSet` call.
- `Ii iiplot` view initialization using a `comgui objSet` call.
- ...

8.1.4 File name generation with `objString`

The ability to generate context based file names is obtained using `comgui objString`. The principle is to provide a cell array of strings where `'@command'` string are interpreted.

8.1.5 Image generation with `ImWrite`

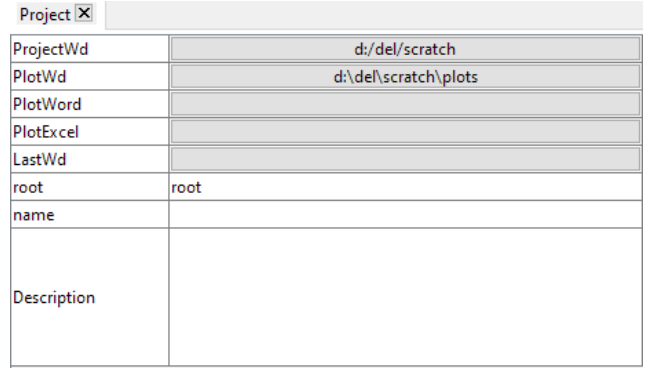
8.2 SDT Tabs

This section presents the GUI of SDT, organized as tabs in the `sdtroot` figure.

- The application tools breakdown is provided in an exploration tree placed at the figure left. The buttons allow opening the corresponding interface tabs.
- The tab area displays interactive tables that allows parameter editing and procedure execution. User interaction is associated with tabs implemented in the GUI,
  - `Project` tab to handle the working environement, section section 8.2.1 .
  - `FEMLink` tab to handle model imports, section section 8.2.2 .
  - `Mode` tab to handle modal computations, section section 8.2.3 .
  - `TestBas` tab to superpose two meshes, section section 8.2.4 .
  - `Ident` tab SDT identification tuning, section section 8.2.6 .
  - `StabD` tab for stabilization diagrams, section section 8.2.5 .
  - `MAC` tab to handle MAC analysis, section section 8.2.7 .
  - `OsDic` tab for `sdtroot OsDic` editing, section section 8.1.3 .

8.2.1 Project

The `Project` tab allows handling the working environment.

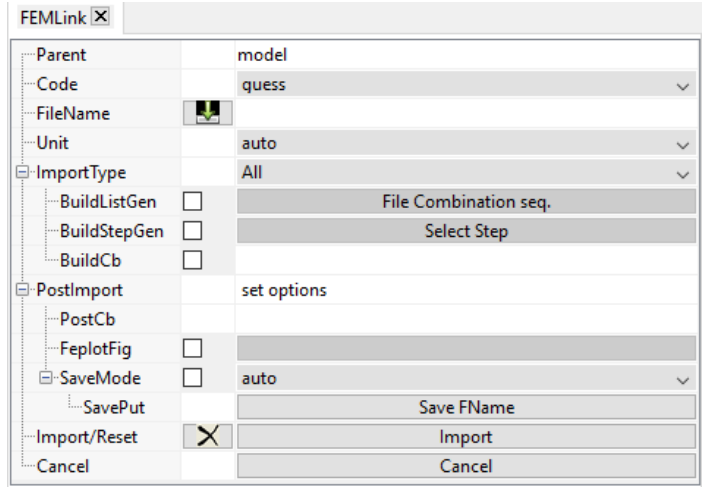


This is a 2 column table allowing the definition of the following fields,

- `ProjectWd` A button defining the working directory used for the project. This is where models and curves will be saved. Clicking on the button will open a dialog for interactive definition.
- `PlotWd` A button defining the directory where image captures will be saved. If not specified the default will be `ProjectWd/plots`. Clicking on the button will open a dialog for interactive definition.
- `PlotWord` A button defining an existing Word report to which captured images can be inserted. Clicking on the button will open a dialog for interactive definition.
- `PlotExcel` This is not currently used, but could allow the specification of a different file for table export.
- `LastWd` The last chosen directory, used as a starting point for the next directory selection dialogs.
- `root` A short name that will be used to identify saved files in the project working directory, every saved file will start with this root.
- `name` A longer name version that is used for human description of the project name.
- `Description` An optionnal text that can provide further details on the current project.

8.2.2 FEMLink

The FEMLink tab allows handling model import from external codes.



This is a three column table allowing interactive definition of the fields described below. The second column allows activating specific options.

- Parent string name used to identify the model in further post-processing operations.
- Code allows selecting the code from which files will be imported. If code is unknown femlink will try guessing it from the file extension. This is a popup button providing a specified list of options. This is set by default to unknown.
- FileName Provides the base file for import. This file will be imported first and constitute the base model for the output. The second column button allows an interactive file selection through a dialog. The third column is an editable text cell.
- Unit allows defining a unit system with the model, that can be used for post treatments where output units are required. Some codes do not use it so that an external defintion is needed. This is set by default to auto.
- ImportType Provides model building options based on complementary files
  - *All* imports model, results, ...
  - *Model* just imports the model, material properties, boundary conditions, ...
  - *Result* import result.
  - *UPCOM SE* import element matrices in a type 3 superelement handled with upcom.
  - BuildListGen allows generating a file list sequentially built, by successive file selection. These files then appear under the BuildListGen button and can be removed from the list

by clicking on . This is illustrated in figure 8.4. This option conditions the activation of BuildStepGen and BuildCb below.

- BuildStepGen : Should be updated with a capture of the window for step selection. Allows defining model Case resolution for a specific results step. By default femlink imports all data in the model. To recover specific boundary conditions relative to a specific computation step (if defined in the input and supported by the femlink function), one can either provide the step number or ask for last to let femlink find the last step defined in the model load case. The third column button allows selecting a step in an interactive way. By default, this option not is activated.
- BuildCb Allows defining further Build commands that may depend on the Original Code selected. The second column activates the option. The third column button provides a series of comma separated calls that will be applied to the model generated by femlink through the FEMLink function defined by the Code.

- PostImport is used to define steps performed after the base import.
  - PostCb callback performed after import (for custom applications using FEMLink).
  - FeplotFig Allows direct model loading into a feplot figure for visualization. The second column button activaftges the option. The third column button allows interactive selection a feplot figure, like for the Project tab. By default, this option is not activated.
  - Save allows defining model saving strategy once imported. The second column button activates the option. The third column button allows defining a saving mode. This is a popup button proposing either :
    * *auto* that will perform an automatic saving of the model based on the Mesh File name with a _import.mat extension
    * *Link to Project* that will use the Project tab data to generate a file name. In this case the saving file name will be Project.root _ date _ Mesh File name _ import.mat
    * *Custom fname* allows defining a user specific name in the second line button. The Save FName button can then be clicked on to provide a file name that will be used as verbatim.

    By default the save option is activated and set to Link to Project.

- Import/Reset Import executes the import proecdure. The cross resets the tab to its original state.

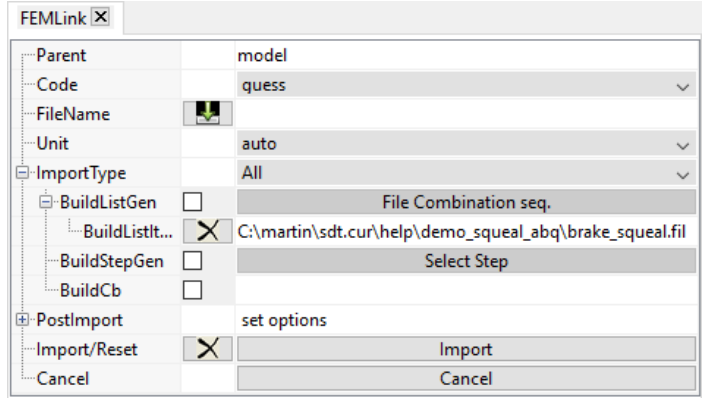
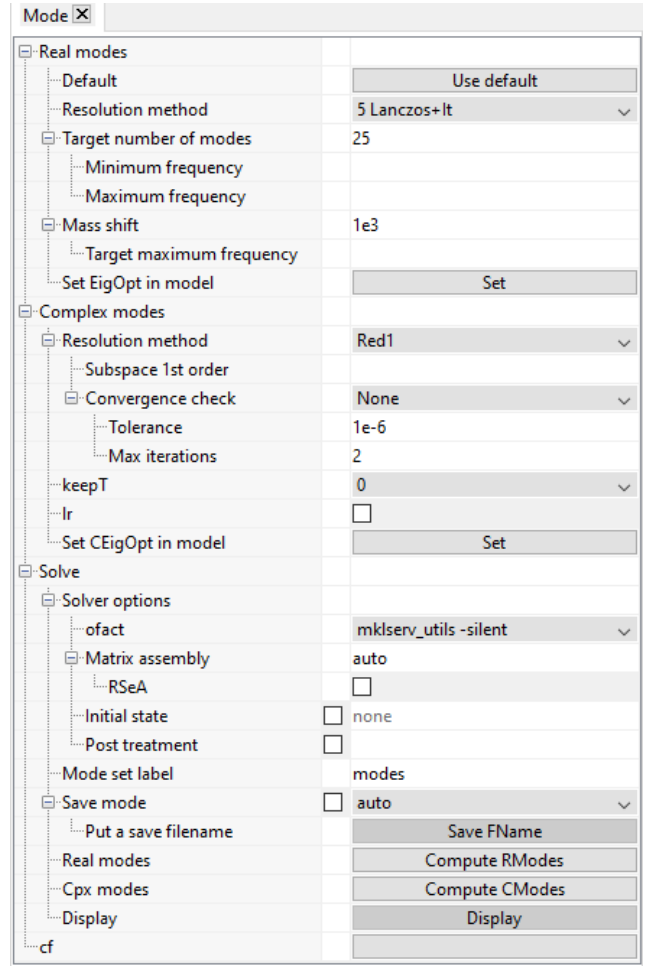


Figure 8.4: The `FEMLink` tab, filled with input files

8.2.3 Mode

The `Mode` tab allows handling modal computations.



This is a three column tree-table allowing various choices to perform a wide range of modal computations, parametered by the fields below,

- `Real modes` This section and the associated subtree provides options on the computation of real modes
  - `Default` Resets parameters of the real mode subtree to default values
  - `Resolution method` The real mode solver (resolution method) choice (also used for reduced complex mode computations). Choices are packaged in a popup cell :
    * *`Lanczos+It`* : set by default and recommended
    * *`Lanczos`* : same as the previous without convergence check and correction, be used once parameters are calibrated

        * *IRA/Sorensent* : quicker but less robust
    - Target number of modes To provide a number of modes to compute, set to 25 by default.
        * Minimum frequency To provide a minimum frequency of interest (not packaged yet).
        * Maximum frequency To provide a maximum frequency of interest.
    - Mass shift To provide a mass shift used for the factorization. This is set to 1e3 by default.
        * Target maximum frequency To provide clues on the expected bandwidth (will influence the mass shift).
    - Set EigOpt in model

- Complex modes This section and the associated subtree provides options on the computation of complex modes
    - Resolution method The complex mode solver (resolution method) choice. Choices are packaged in a popup cell either :
        * *Red1* : complex modes on the real mode subspace (default)
        * *Red2* : complex modes on the real mode subspace enhanced with the imaginary part of the stiffness
        * *Full* : direct without reduction
        * Subspace 1st order The choice of the matrix types to be used for the subspace enhancement. The visc option is only available with SDT-visco licenses.
        * Convergence check Not packaged yet.
            · Tolerence
            · Max iterations
    - keepT
    - lr
    - Set CEigOpt in model

- Solve This section and the associated subtree provides options on the solver to use and potential post treatment or saving strategies.
    - Solver options
        * ofact The choice of the matrix factorization solver, set by default to mklserv_utils -silent. This is recommended for very large models.
        * Matrix assembly A text cell providing the matrix types to be assembled for the computation. This is either the keyword auto to let the solver decide the assembly strategy, or a series of matrix types (see sdtweb mattyp) to be assembled. By default this is set to auto, corresponding to 2 1 for real modes and 2 3 1 4 for complex modes.

· RSeA

* Initial state This is activated by the second column. The third column provides a callback to initialize the system state. (not packaged yet).
* Post treatment Allows performing a callback after mode computation. The second column activates the option. The third column is a text cell providing a callback to perform. Not packaged yet.

– Mode set label A curve name used to store the deformation curve in the model stack. This is a text cell, set by default to modes.

– Save mode allows automatic curve saving once imported. The second column button activates the option. The third column button allows defining a saving mode. This is a popup button proposing either :

* *auto* that will perform an automatic saving of the curve based on the base model name with a _def.mat extension.
* *Link to Project* that will use the Project tab data to generate a file name. In this case the saving file name will be Project.root _ date _ model name _ Mode set label _ def.mat
* *Custom fname* allows defining a user specific name in the second line button.
* Put a save filename The Save FName button can then be clicked on to provide a file name that will be used as verbatim.

– Real modes Executes the real mode computation

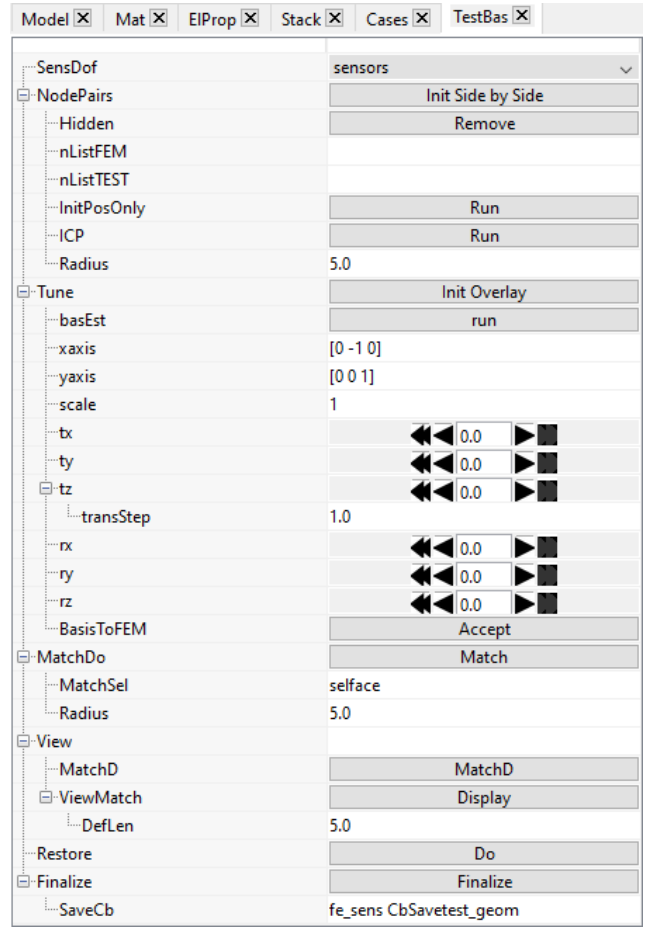
– Cpx modes Executes the complex mode computation

– Display Displays the model stack entry named after Mode set label.

- cf A button allowing an interactive defintion of the feplot figure that will hold the working model. Clicking on the button opens a dialog interface proposing the selection of an existing feplot figure or to open a new one. By default, this is set to the one specified in the Project tab.

8.2.4 TestBas : position test versus FEM

The TestBas tab is used to superpose two meshes. For examples see section 3.1 .



This is a tree-table used for mesh superposition. The base mesh is called FEM and the mesh to be placed is called TEST even when you are superposing different things (TEST/TEST, FEM,FEM, ...). The `NodePair` section uses a strategy providing corresponding points, while the `Tune` section allows manual tuning of the relative position.

- `SensDof` selects the second mesh (stored in as a `SensDof` case entry in the first mesh) to be superposed on the first one.
- `NodePairs` is used to initialize the `FemTest` dock in side by side mode. In this mode, the left tile shows the `feplot promodel`, the center tile shows the reference mesh and the right tile the `SensDof` mesh.

  The first step in this mode is to provide two list of paired nodes for the two meshes. To select the nodes, select a `feplot` figure and press the `space bar`: clicking on the mesh will select

nodes and add them to the list. Doing so in the two feplot figures provides two sets on paired nodes that can be used superpose with this information only (InitPosOnly) or with the help of an automatic algorithm after the initial positioning (ICP : Iterative Closest Points).

  - Hidden To ease selecting only visible nodes on each mesh, this button removes hidden elements from the camera point on view. (Useful when selecting the paired node lists)
  - nListFEM list of nodes selected in the main mesh feplot in center tile.
  - nListTEST list of nodes in SensDOF mesh (right tile).
  - InitPosOnly superpose the two meshes by minimization the Euclidian distance between the previously filled lists of paired nodes. This is helpful in presence of geometries with symmetries for which the ICP algorithm cannot converge (plate or cylinder for instance).
  - ICP, using paired node lists, performs first the InitPosOnly action and then starts the optimization with the algorithm ICP which seeks to minimize the point-to-plane distance between each automatically paired nodes (closest nodes in the range of Radius).
  - Radius search radius for node pairing (this is the same value as the Radius in MatchDo)

- Tune opens the FemTest dock inTune mode. The left timeshows the feplot promodel while the right feplot overlays the reference mesh (in blue) and the test mesh in its current position (in red)

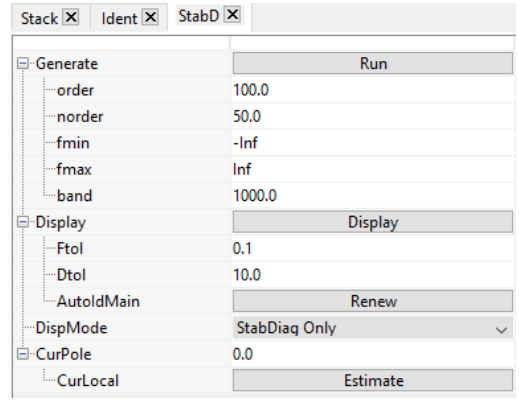
  - basEst : starting guess : if no InitPosOnly has been performed, the two meshes are automatically superposed using the gravity center and the three main directions of the point clouds formed by each node mesh. This is helpful to be closer to the good superposition before beginning to tune manually
  - xaxis This is an informative display which gives the orientation of the x-axis test coordinate system in the base model. This is updated when rotating the second mesh
  - yaxis orientation of test y-axis in FEM coordinates.
  - scale scale applied between the two coordinate systems (for FEM in mm and test in meters use 0.001).
  - tx Translation of test in the x-direction. The single arrows correspond to a low displacement step and the double arrow to a higher displacement step
  - ty Translation of test in the y-direction
  - tz Translation of test in the z-direction
    * transStep This is the translation step used by the single arrow.
  - rx Rotation of the second mesh around the x-axis. This rotation does not increase the angle which is always zero, but updates the orientation of the xaxis and the yaxis. The text is used for using input of large changes 90 (degrees) for example.

  - ry Rotation of the second mesh around the y-axis
  - rz Rotation of the second mesh around the z-axis
  - BasisToFEM Modify the SensDof mesh by applying the transformation. The node coordinates are modified and all Tune fields set to identity.

- MatchDo Match is automatically performed after ICPPosOnly, ICP and BasisToFEM. This button can be used to redo the match with new options below.
  - MatchSel Selection on the FEM before performing the Match. selface is classically used to force the match on the surface of the model instead of in the volume.
  - Radius Search radius for node pairing (this is the same value as the Radius in NodePairs)
- View List of different views to evaluate the quality of the superposition
  - MatchD displays the table showing the gap between each node of the second mesh and the matched surface. It also shows this information as a colormap on the test.
  - ViewMatch Displays the test mesh over the FEM with the options listed below
    * DefLen Length of arrows if displayed
- Restore uses the .bas0 field to reset all the modifications since the last BasisToFem (performed after clicking on InitPosOnly, ICP and BasisToFEM) and put the second mesh at this previous location.
- Finalize Performs the SensMatch (i.e. the observation of the first mesh at sensors)
  - SaveCb Callback executed with the Finalize action

8.2.5 StabD : stabilization diagram

The StabD tab is used create a stabilization diagram with the algorithm LSCF and provide tools to extract poles from it.

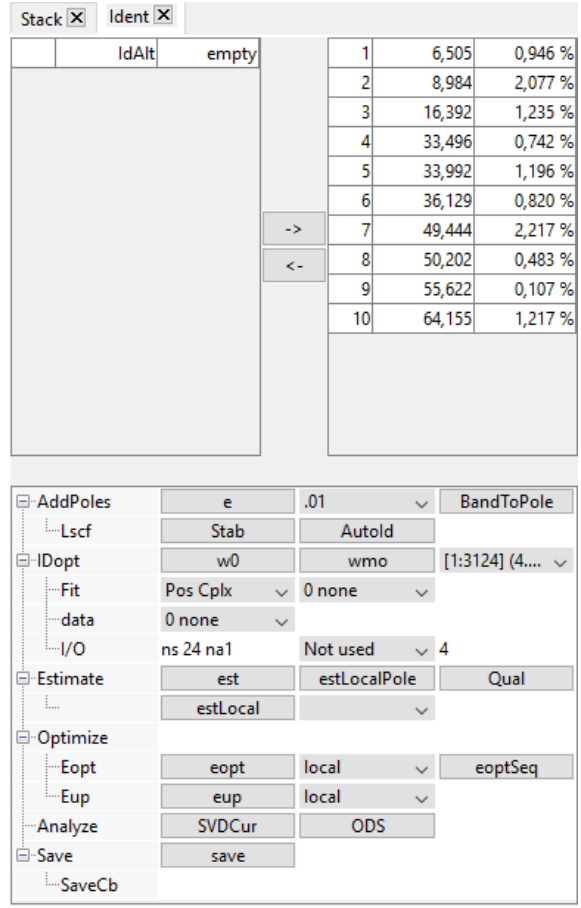


This is tab is used for LSCF handling (see section 2.3.2).

- `Generate` click on button to generate stabilization diagram.
  - `order` : Maximum order of the model. The order of the model equals the number of poles used to fit the measured data. It is often necessary to select an order significantly higher than the expected number of physical poles in the band because the identification results in many numerical poles which compensate out-of-band modes and noise. Selecting at least ten times the number of expected poles often gives good results according to our experiment.
  - `norder` : Minimum order to start the stabilization diagram (low model orders often show very few stabilized poles)
  - `fmin` : Minimum frequency defining the beginning of the band of interest
  - `fmax` : Maximum frequency defining the end of the band of interest
  - `band` : Sequential iteration can be performed by band of the specified frequency width. The interest is that in presence of many modes, it is more efficient to perform several identifications by band rather than increasing the model order.
- `Display` : display result
  - `Ftol` : tolerance for frequency convergence
  - `Dtol` : tolerance for damping convergence
  - `AutomIdMain` : fill `IdMain` set of poles from current data.
- `DispMode`
- `CurPole` : info based on click.
  - `CurLocal`

8.2.6 Ident : pole tuning

The `TabIdent` tab is



The upper part is a list of alternate poles on the left and retained poles on the right. The arrows let you move poles and associated shapes from one list to the other.

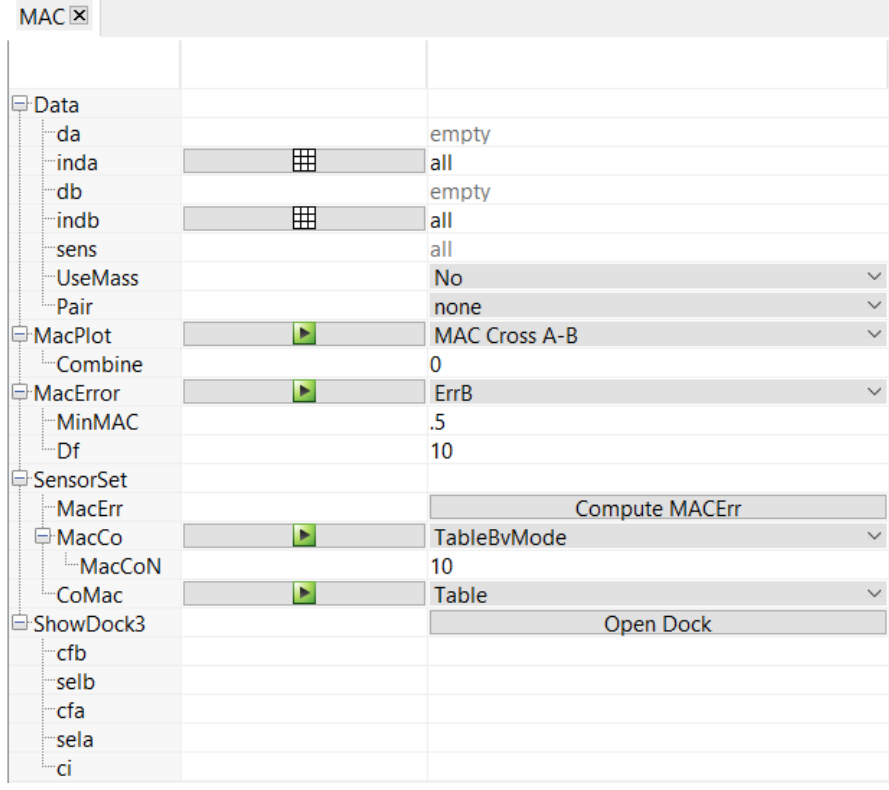
The lower part as the main sections

- `AddPoles` see section 2.3
  - `Lscf` LSCF algorithm see section 2.3.2
- `IDopt` section 2.4
  - `Fit`

  - data
  - I/O
- Estimate section 2.5
- Optimize section 2.6
  - Eup
- Analyze section 2.8
- Save
  - SaveCb allows customization of saving strategy

8.2.7 MAC : Modal Assurance Criterion display

The MAC tab allows handling display of variants of Modal Assurance Criterion.



This is a three column tree-table allowing various choices to perform a wide range Modal Assurance Criterion variants, parametered by the fields below

- Data Options to properly define input data
  - da provides indications on the number of sensors and the number of modes of
  - inda
  - db
  - indb
  - sens
  - UseMass
  - Pair
- MacPlot
  - Combine
- MacError
  - MinMAC
  - Df
- SensorSet
  - MacCo
    - MacCoN
  - CoMac
- ShowDock3
  - cfb
  - selb
  - cfa
  - sela
  - ci

8.3 Non java SDT Tabs (obsolete)

In the obsolete non java rendering mode (`iicom(';projava0;curtabChannel')`) the channel tab shown below is used to scan through the data.

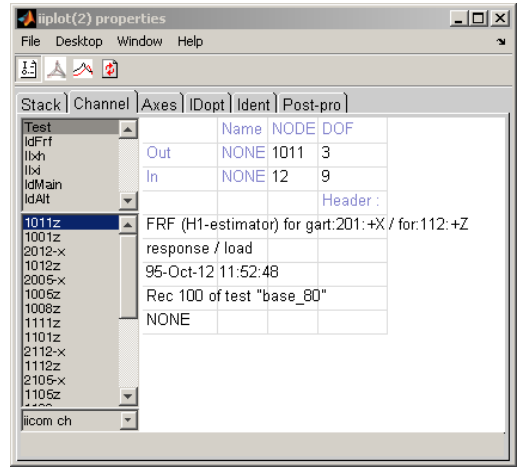


Figure 8.5: Channel tabs of the `iiplot` interface.

Major commands you might want to know

- 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 `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')`.

8.4 Handling data in the GUI format

8.4.1 Parameter/button structure

The initialization of GUI button/cells is performed using a `but` structure with fields `.type`, `.name`, ... Available types (string in `but.type`) are

- `string` A free input as a string or a number
- `pop` An input chosen in a predefined list
- `push` An assisted input triggered with a click on the button, or an action to execute
- `check` An on/off input, that can be equivalent to `pop` with two entries, but in a checkbox shape rather than a list

Each parameter or action is thus associated to a button of the types presented above. The parameter definition then depends on the type, as presented.

- For a `string` type, following fields are accessible and mandatory (if not stated otherwise)
  - `name` The button name, explicited as `family.param`, that defines the parameter and its accessibility.
  - `format` The data format, either `%s` for a real string or `%g` for numeric (double) values.
  - `value` The parameter current value.
  - `enable` (Optional) A logical, or either `'on'` or `'off'`. That allows deactivation of the parameter edition.
  - `ToolTip` A string briefly defining the parameter.
  - `SetFcn` A function to be called if dependencies have to be handled after editing the parameter. This can be left as an empty string (`''`). Field `.name` is necessary and field `.parent` may be needed to access the containing table.
  - `type` A fixed string defining the parameter type, here `string`.
  - `ContextMenu` a JPopupMenu that will be active in Java rendering of the cell. This field applies to all types.
- For a `pop` type, following fields are accessible and mandatory (if not stated otherwise)
  - `name` The button name, as `family.param`, that defines the parameter and its accessibility.
  - `choices` A cell array defining the choices available to the user. All choices are strings.
  - `choicesTag` (Optional) A cell array defining the choices available to the user. All choices are strings. For localization matters, the language displayed in field `choices` may vary. This entry is thus a constant cell generally corresponding to the coding language. It is then possible to test the choice string parameter in the code with a fixed language independently from the display.
  - `value` An integer providing the current choice.

- `enable` (Optional) A logical, or either `'on'` or `'off'`. That allows deactivation of the parameter edition.
- `ToolTip` A string briefly defining the parameter.
- `SetFcn` A function to be called if dependencies have to be handled after editing the parameter. This can be left as an empty string (`''`). A `name` must be defined.
- `type` A fixed string defining the parameter type, here `pop`.

- For a `push` type, following fields are accessible and mandatory (if not stated otherwise)
  - `name` The button name, given as `family.param`, that defines the parameter and its accessibility.
  - `value` A string containing the parameter value, or the action name to be displayed.
  - `enable` (Optional) A logical, or either `'on'` or `'off'`. That allows deactivation of the parameter edition.
  - `ToolTip` A string briefly defining the parameter.
  - `callback` A function to be executed when triggering the edition.
  - `SetFcn` *not normaly used*. Since push cannot be edited, no dependencies can occur.
  - `type` A fixed string defining the parameter type, here `push`.
- For a `check` type, following fields are accessible and mandatory (if not stated otherwise)
  - `name` The button name, given as `family.param`, that defines the parameter and its accessibility.
  - `value` An integer being `0` or `1` depending on the parameter state.
  - `enable` (Optional) A logical, or either `'on'` or `'off'`. That allows deactivation of the parameter edition.
  - `ToolTip` A string briefly defining the parameter.
  - `SetFcn` A function to be called if dependencies have to be handled after editing the parameter. This can be left as an empty string (`''`).
  - `type` A fixed string defining the parameter type, here `check`.

8.4.2 DefBut : parameter/button defaults

To ease the development of GUIs, buttons are stored in `DefBut` structures. Initialization of the `DefBut` is usually done in using a file see section 8.4.3 .

`DefBut.MyField` will usually group all buttons needed for a given part of the interface. Notable exceptions are

- `.Tab` used to store information associated with floating tabs. In particular `.Tab.(field).jProp` stores properties for java initialization.
  `.InitFcn={'fun','command'}`. `.SetFcn={'fun','command'}`.
- `.j` used to store volatile java objects that should not be reinitialized too often.
- `.fmt` is a cell array containing the `OsDic` style sheet (text keys in first column and values in second).

The set of parameters is divided into families and defined by a keyword and a type. Each family can be easily displayed in separated tabs of the GUI, and constitute relevant sets of parameters regarding human readability.

For generalization purposes, execution actions follow the same definition as parameters, and are linked to a family, keyword and type.

The families and keywords are left free as long as they are compatible with the definition of MATLAB `struct` fields. The parameter type allows defining which kind of action the user is provided for edition. This is realized in the display by adapted buttons.

Each parameter can be defined as a structure, nested in a structure containing the parameter families as fields. The generation of such structure is handled by `sdt_locale` so that the definition consists in the generation of a `csv` file in ASCII format.

8.4.3 Reference button file in CSV format

The input `csv` file layout allows defining a parameter, or button with a header line starting with `h;` defining its type and the fields to be provided, and an instance line starting with `n;` providing the fields value. Fields that are invariant for the whole class can be defined in the header line. Comments are possible with lines starting with `c;`.

The following example illustrates the definition of each type of buttons

```
c; Sample definition of each class

c; sample string buttons, with dependencies handled by function my_ui
h;type=string;name;format;value;ToolTip;SetFcn=''
n;Family.SampleStrS;%s;"st1";"a string input button with no dependencies"
n;Family.SampleStrG;%g;1;"a numeric input button with no dependencies"

c; sample pop button
h;type=pop;name;value;choices;choicesTag;ToolTip;SetFcn=''
```

```
n;Familiy.SamplePop;1;{'choice1','choice2'};{'c1','c2'};"2 choice menu with default choice1"

c; sample push button
h;type=push;name;callback;value;ToolTip;
n;Family.SamplePush;my_fun('exec');"Push this button";"push button triggering my_fun"

c; sample check button
h;type=check;name;value;enable;ToolTip;SetFcn=''
n;Famimty.SampleCheck;0;"on";"check button, set 0, with conditional enabling and no dependencies"
```

The csv file should be named after the GUI handling function my_ui, a standard language identifier and extension .csv. Here for example my_ui_en-us.cvs for english-US or my_ui_fr-fr.cvs for french.

Generation of the parameter structure classically named DefBut can then be obtained by

```
DefBut=sdt_locale('defCSV','my_ui_en-us.csv');
```

At this state of definition, DefBut is a standard MATLAB struct corresponding to the documented fields. To transform it into a java object linked to the GUI figure of handle GuiGF, command cinguj ObjEditJ must be used

```
[r1j,r1]=cinguj('objEditJ',DefBut.Family,GuiGF);
```

The first output is r1j, which is an EditT Java object. This object will be used for dependencies handling and can be edited using sdcedit. The second output r1 contains copies of each parameters in a struct with fields the parameter names. The parameters are in their Java form that is to say editable buttons of class CinCell.

8.4.4 Data storage and access

Initializing the GUI figure

After generating the Java objects containing the parameters, one can store them in the GUI figure for further access. The data are stored in the GUI figure that is initialized by cinguj ObjFigInit.

```
GuiGF=cinguj('objFigInit',...
struct('tag','my_ui','noMenu',1,'name','my_ui'));
```

The handle should be stored UI.gf field of persistent variable UI in my_ui. One can also recover this pointer at any time by using GuiGF=findall(0,'tag','my_ui'). It is thus critical to ensure the unicity of the GUI figure tag.

Efficient data storage in a figure is handled in SDT through the use of v_handle uo object. Access to this pointer is possible at any time using

```
uo = v_handle('uo',GuiGF);
```

It is recommended to package the access to the java data pointer in a command uo=my_ui('vh').

Handling the data java pointer

Automatic storage of the data pointer is performed at display. The pointer is handled as a MATLAB struct with fields corresponding to the parameter families. The objects stored are then either the EditT containing the full parameter family or a struct of CinCell, respectively corresponding to the first and second outputs of the ObjEditJ command.

A very low level way of storing invisible data is to edit the uo object directly by doing

```
r1=get(GuiGF,'UserData');
r1.(family)=r1j;
set(GuiGF,'UserData',[],'UserData',r1);
```

where family is the parameter family, r1j the EditT object generated by ObjEditJ and GuiGF the handle to the GUI figure. It is however recommended to let it be stored automatically at display.

Recovering data from java objects

To recover data in a RunOpt MATLAB struct format from EditT or CinCell objects, command fe_defCleanEntry must be used.

- For an EditT object the output of CleanEntry will be a structure with as many fields as parameters stored in the EditT assigned with their value converted to the proper format provided. When an EditT is displayed in a tab, obj.Peer should be the numeric handle to the matlab figure so that clean_get_uf can retrieve tab data.
- For a CinCell object, the output of CleanEntry will be the underlying structure of the button, as documented. Each CinCell object can/should have a EditT parent obtained with obj.get('parent').

- For pop objects `CinCell` or `struct`, the value is taken to be the `choicesTag` string if it exists or the `choices` string otherwise.

`fe_defCleanEntry` no longer returns the full structure for a button, so that the command `r1=cinguj('ObjToStruct',ob);` should be used.

To get the current data (`.data{.val}` of pop button, one uses `r1=feval(sdtroot('@obGet'),ob,'data');`.

It is recommended to build a call `my_ui('GetTab')` that will rethrow the `RunOpt` structure corresponding to a `Tab` from the GUI figure.

```
% get Java pointer and desired tab field
out=my_ui('vh'); tab=varargin{carg}; carg=carg+1;
% convert to a RunOpt structure
out=fe_def('cleanentry',out.(tab));
```

Direct access to a parameter can also be usefully packaged in `my_ui('GetTab.Param')`, with

```
% get Java pointer and desired tab field
out=my_ui('vh'); tab=varargin{carg}; carg=carg+1;
% parse tab to see if subfields are desired
tab=textscan(tab,'%s','Delimiter','.'); tab=tab{1};
% convert to a RunOpt structure
out=fe_def('cleanentry',out.(tab{1}));
% output only the desired subfield if it was specified
if length(tab)>1; tab(1)=[];
 while ~isempty(tab)&&~isempty(out); out=out.(tab{1}); st(1)=[]; end
end
```

Displaying data in the GUI figure

To display the parameters in the GUI figure, one has to generate a structure that will be interpreted as a `JTable` that will be included to the `JTabbedPane` object, that is to say the tabbed area of the GUI figure. This structure contains the fields

- `name` The name of the object that will be display. It is recommended to use the family name of the parameter family displayed.
- `table` A `cell` array containing the buttons in the `CinCell`. The `JTable` will have the same size as the table provided.

- ToolTip A string allowing to display some explanations on the tab.
- ParentPanel The handle to the GUI figure.

Generation of the table field can be done automatically with a call to sdt_dialogs uatable

```
ua.table=sdt_dialogs('uatable-end0','info',name,r1j);
```

with name the field relative to ua.name and r1j the EditT object (with .Peer defined). This will yield a tab with three columns, the first one being the parameter names, the second one the editable buttons as CinCell objects and the third one being the parameter ToolTip.

More complex layouts can be obtained by generating the table manually, exploiting the second output of the ObjEditJ command to fill in table positions. This allows generating the table by directly positioning the CinCell objects called by their names.

By adding a field level to ua, and calling cingujTabbedPaneAddTree a tree will be displayed instead of a table in the GUI figure. Field level has two columns and as many lines as the table. The first column provides the level of the table line in the tree as an integer. The second column indicates whether the line has to be expanded is set to 1, or not if set to 0.

Once ua is filled display is performed using cinguj TabbedPaneAdd

```
[ua,ga]=cinguj('TabbedPaneAdd','my_ui',ua);
```

Command TabbedPaneAdd outputs ua that contains the displayed objects and their information. This can be accessed any time using field tStack of the GUI figure userdata, `uf=clean_get_uf(GuiGF)`, and ga that is the handle to the figure axis containing the tab.

8.4.5 Tweaking display

Display can be tuned to the user will by editing the displayed objects. All display information is accessed through a call to clean_get_uf, using GuiGF the GUI figure handle as input argument.

```
uf = clean_get_uf(GuiGF);
```

uf is a user data structure with fields

- ParentFigure The GUI figure handle. This should be equal to GuiGF.
- p The handle to the uipannel displaying the data.
- tStack A cell array of 7 columns and as many lines as tabs generated. Column 1 contains the tab names and column 7 contains the tab userdata object.

- `tab` the index in `tStack` corresponding to the tab currently displayed.
- `java` set to 1. Ensures that the userdata handles java objects for `cingui`.
- `JPeer` A pointer to the Java object containing the display, either a `JTabbedPane` if only tabs are displayed, or a `JScrollPane` if only a tree is displayed, or a `JSplitPane` if the display contains several Panes.
- `pcontainer` The handle to the `hgjavacomponent` that contains the display.
- `toolbarRefresh` (Optional) A function handle that can be called at refresh to perform toolbar dependencies (e.g. `uicontrol` enabling as function of the GUI state.
- `tag` The GUI figure tag.
- `Explo` If an exploration tree is present, the `JScrollPane` java object containing the tree.
- `EJPeer` If an exploration tree is present, `JSplitPane` java object containing the global display.

The seventh column of `uf.tStack` contains information relative to each of the tab objects of the `JTabbedPane`. It is commonly named `ub` and contains the following fields

- `name` The tab name, that should be corresponding to the parameter family.
- `table` A cell array containing the objects of each cell of the `JTable`
- `ToolTip` A string providing a tool tip if the mouse cursor if over the tab tip.
- `ParentPannel` The handle to the GUI figure.
- `type` A string providing the table objects type, commonly `CinCell`.
- `JTable` The JTable java object.
- `JPeer` pointer to the JScrollPane typically used for display.
- `NeedClose` value set to force use of a close button on the tab.

Each tabbed pane can be tweaked regarding the displayed column dimensions.

In the case of a GUI displaying user input objects the table itself does not need to be interactive. (This is different from a results table that will be analyzed by the user). It is thus recommended to deactivate the table selection interactivity using

```
ub.JTable.setRowSelectionAllowed(false);
ub.JTable.setColumnSelectionAllowed(false);
ub.JTable.setCellSelectionEnabled(false);
```

Columns width can be set using a line array with as many columns as columns in the table and providing in pixels the minimal width a column should have to cingujtableColWidth. The value can be set to -1 if the user wishes to let free the width of a column.

```
% for 3 columns table, last one left free
ColWidth=[150 300 -1];
cinguj('tableColWidth',ub.JTable,ColWidth);
```

Row height can be set (same for all lines) by calling the setRowHeight method of JTable. The value is in pixel.

```
% getting the intial row height
r1 = ub.JTable.getRowHeight
% setting a new row height to 22px
ub.JTable.setRowHeight(22)
```

8.4.6 Defining an exploration tree

To ease up navigation between tabs, one can use an exploration tree in the GUI figure. Tabs can then be opened by clicking in the tree that should list all available tabs (or parameter families).

The exploration tree is commonly named PTree, and has to be defined in the .csv file. It should contain push type buttons with callbacks triggering the opening of the desired tab.

```
c; sample PTree defintion
h;type=push;name;callback;value;ToolTip;
n;PTree.Family;my_ui('InitFamily');"Family";"Open corresponding family tab"
```

To properly handle an exploration tree, one has to initialize it when the GuiGF figure is opened, that is to say after the cinguj ObjFigInit call. The initialization should be handled by a call of the type mu_ui('InitPTree').

Low level access to the exploration tree is handled by a subfunction of cinguj named treeF. The subfunction handle can be accessed using treeF=cinguj('@treeF');. It is recommended to store the variable treeF containing the subfunction handle in a persistent variable of the GUI function my_ui.

```
% option initialization
RunOpt=struct('NoInit',0,'lastname','');
% for all fields of DefBut.PTree, sort the buttons
r1=fieldnames(DefBut.PTree); table=cell(length(r1),2);
for j1=1:length(r1);table(j1,:)={DefBut.PTree.(r1{j1}) [1 1]};  end
```

```
% generate clean table and corresponding levels
level=vertcat(table{:,2});  table=table(:,1);
% generate the tree ua
ua=struct('table',{table},'level',level,'name','my_ui',...
 'ParentPanel',GuiGF,'ToolTip','The GUI exploration tree','NeedClose',2);
% display the tree in the GUI figure
[tree,gf]=cinguj('tabbedpaneAddTree','my_ui',ua);
% tweak the tree to enable selected tab field highlighting
tree.getSelectionModel.setSelectionMode( ...
 javax.swing.tree.TreeSelectionModel.SINGLE_TREE_SELECTION)
% refresh
 cingui('resize',GuiGF);
```

The exploration tree thus defined highlights its node corresponding to the currently displayed tab. This tasks is performed automatically by `cinguj` when clicking on a button of a tree.

To access the tree object and its highlighted field, one can do

```
 [RunOpt.lastname,tree]=treeF('explolastname',GuiGF);
```

To switch the highlighted field to a new name `newname and get the tree node object`, one can do

```
 node=treeF('scrollToNameSelect',tree,newname);
```

8.4.7 Finding `CinCell` buttons in the GUI with `getCell`

To quickly find `CinCell` buttons in an interface, subfunction `getCell` of `sdcedit` can be used.

```
getCell=sdcedit('@getCell');
[obj,tab,name]=getCell(r1j,'propi','vali',...,stOpt)
```

- `rj1` is a GuiGF, or an UIVH, or a java/EdiT, or figure Tag, or vector of handles or 0 for all MATLAB figures.
- `propi, vali` are pairs of properties (fields of the buttons) and their desired values.
- `stOpt` is an option that allows a constant output in cell format if set to `'cell'`.

Actions to check or get specific fields of a cell array of `CinCell` buttons are also available using commands

```
% r1=getCell('getfield st',obj); % outputs field st of obj (CinCell) or {obj}
% r1=getCell('isfield st',obj); % outputs logical checking presence of field st in obj
```

8.5 Interactivity

8.5.1 Handling tabs

To initialize tabs, it is recommended to use a call of type `my_ui('InitTab')`, that handles the tab generation using the standard button definitions.

To get information on the existing tabs, one can access to `uf`, with `clean_get_uf`.

It is possible to switch the display to an existing tab using `cinguicurtab`*`Tab`* command, with *`Tab`* the tab name to switch to.

To close a tab, one should use a call to subfunction `tabChage` of `cinguj`. Handle to the subfunction can be accessed with `cinguj('@tabChange')`. One must then provide the `close` command, the GUI figure tag, and the tab name to close.

- One can use *`_cur`* instead of a tab name to close the current tab.
- One can use command `closeAll` instead of `close` to close all tabs at once.

```
% close current tab:
feval(cinguj('@tabChange'),'close','my_ui','_cur')
% close tab 'tab'
feval(cinguj('@tabChange'),'close','my_ui',tab)
% close all tabs
feval(cinguj('@tabChange'),'closeAll','my_ui')
```

8.5.2 Handling dependencies

Dependencies define the set of actions performed consequently to the edition of a given parameter. They should be handled by a call of type `my_ui('set')`. Classically dependencies are handled through the `SetFcn` definition relative to each parameter. In the `.csv` definition, most `SetFcn` fields should be set to `my_ui('set')`.

For the exclusive case of `push` buttons, dependencies or actions have to be passed to the `callback` field.

The set function call must be able to be called from script in the same manner than from CinCell callbacks. Calls of the form `my_ui('set',struct('Tab.Par',val,...));` should then edit the parameters and execute dependencies.

A typical entry to the set command can then be

```
if carg<=nargin; % from script mode
 r1=varargin{carg}; carg=carg+1; r2=fieldnames(r1);
 if length(r2)>1 % allow multiple fields input at once
   for j1=1:length(r2); % loop to assign each field
    my_ui('Set',struct(r2{j1},r1.(r2{j1})));
   end
   return % get out after having assigned each parameter
  else % one parameter provided, carry on
   obj=r1j.(CAM).(r2{1}); val=r1.(r2{1}); gf=GuiGF;
   uo=struct('FromScript',1); % build the data
  end

 else % callback from CinCell
  [RO,uo,CAM,Cam]=clean_get_uf('getuo',['SetStruct' CAM]);
  obj=uo.ob; val=fieldnames(RO); gf=GuiGF; val=RO.(val{1});
 end
 % robustness check regarding object existence
 if isempty(obj)
  r1=fieldnames(r1); r1=r1{1};
  sdtw('_nb','Property %s does not exist in %s, skipped',r1,CAM); return
 else; CAM=sdcedit(obj,'_get','name'); Cam=lower(CAM);
 end
```

A robust recuperation of the active CinCell is performed through a clean_get_ufgetuo call. Recuperation of the parameter name can be performed with a sdcedit call. Note that obj should be an EditT java object or a CinCell.

To edit or get parameter values it is recommended to use sdcedit that implements robust parameter assignations.

To get values, if the object is an EditT, one should use 4 argument calls of the type `r1 = sdcedit(obj,'field','_get','prop')`, with field the parameter name, and prop the property to get, which is one of the fields defined in the button. A shortcut command to get the property value can be used used `r1=sdcedit(obj,'_get','field')`.

If the object is a CinCell, one can use direct get commands with `r1=obj.get('prop');`.

In the case of pop buttons, the current value can be expressed either as the index in the choices list (or ChoicesTag if defined) or the value in the choices list directly. To ensure the type of data accessed, one can use `st1 = sdcedit(r1j,'field','_popvalue',[])` to get the value in the choice list, or `i1=sdcedit(r1j,'field','_popindex',[])` to get the index in the choice list.

To assign properties, if the object is an EditT, one can use 4 argument calls of the type `r1 = sdcedit(obj,'field',''prop',value)`, with field the parameter name, and prop the property to set to value. A shortcut command to set the property value to val for both EditT and CinCell objects can be used `obj=sdcedit(obj,'field',val)`.

If the object is a CinCell, one can use direct set commands with `r1=obj.set('prop',val);`.

8.5.3 Dialogs

Interaction through dialog windows is possible, and standard sdt_dialogs calls are accessible. Specific dialogs using java objects with interactivity is also possible, but the dialog figure should always be the same and be closed after the dialog to control the number of opened figures.

File input dialog

The most classical dialog is to ask for a file or directory input. If the input file is a parameter in a push button, the user input is handled using a callback with sdt_dialogsEEdit, and the field SetFcn to handle the dependencies.

One can thus define such interactivity with a csv definition like

```
h;type=push;name;callback;value;ToolTip;SetFcn=my_ui('Set')
n;Familiy.FileInput;sdt_dialogs('EEdit_File');"Click to input file";"Specify a file"
```

Call EEdit of sdt_dialogs has several variants,

- EEdit_File to ask for an existing file.
- EEdit_Dir to ask for a directory.
- EEdit_FPut to ask for a file that can possibly be created.
- EEdit_prompt to ask for an input defined through a set of parameters defined in a PropertyUnitTypeCell format.

  ```
  sdt_dialogs('EEdit_prompt m_elastic 2');
  sdt_dialogs('EEdit_prompt -eval"my_fun(''proptypecell'');"',indRequired,val);
  ```

Selection in a tree dialog

When performing design of experiment analyses with saved results, one can use a tree representation of the parameter grid using fe_defRangeTree with a standard SDT parameter structure.

It is possible to implement callbacks in the tree to trigger actions for a specific point, such as loading the selected data set or displaying the selected results.

Using a standard SDT RangeGrid structure here named par, one can display in a dialog figure named my_uidlg the RangeTree that will call a specific loading function with

```
gf=cinguj('ObjFigInit',struct('Tag','my_uidlg,'name',my_uidlg','noMenu',1));
ua=fe_def(['rangetree-outgf-minName-root"DOE"-push-getUA"my_uidlg"'...
     '-callback"sprintf(''my_ui(''''load'''',%%i);'',RO.valLink(j1));"'],par);
cinguj('tabbedpaneAddTree',ua.ParentPanel,ua,'my_uidlg');
```

In the fe_defRangeTree call,

- -outgf Activated the display mode of RangeTree that will generate the java tree object.
- -minName Asks to generate node names to display only with the sub name corresponding to the node level.
- -root''*st*'' Allows specifying a tab root name in the my_uidlg figure.
- -push Activates the generation of push buttons with callbacks for the tree nodes. By default the callback displays in the command window the index in the Range.val list corresponding to the clicked point.
- -getUA''*tag*'' Asks to output the tree object for customized external display. *tag* allows specifying to which parent figure the tree will be displayed.
- -callback''*fcn*'' Allows defining a customized callback. The *fcn* input must in fact be a string that will be evaluated to generate the callback call itself, so that the user can exploit the index in the Range.val list corresponding to the clicked node. Since the displayed nodes are not in the same order than the initial list, fe_defRangeTree uses the internal variable RO.valLink to make the conversion between the displayed node order and the initial val list. The callbacks are generated in a loop in the node order, indexed by j1. In the example, the callback to be generated is my_ui('load',i1) with input i1 being the index of the clicked node in the initial val list.

For such behavior to be relevant, one expects the Range variable par to be accessible at any time by the function my_ui. Saving par in the GUI file arborescence or making it a persistent variable of my_ui are to easy solutions to this issue.

This mechanism can be used to handle a project results file arborescence. In this case the DefBut variable should contain the Range structure that will have to be incremented on the fly when saving a file.

Check list dialog

sdt_dialogs provides a check list functionality handling based on keywords. The associated button is valued as the list of keywords separated with commas. Its edition is then based on a list to check.

The following code demonstrates the use of such button, through a complete definition

```
% Use of buttons associated to a check list
% Define a tab with simple DefBut
r1=['Post(","#push#"define post list")']; % DefBut
% interpet DefBut
R1=cingui('paramedit',r1);
% Specifc check list definition
R1.Post.callback={'sdt_dialogs','EEdit_CheckList'}; % callback
% associate a keyword list
R1.Post.list=['FcA(#3#"Fc stats unfiltered")'...
 'Fc20(#3#"Fc stats filtered 20Hz")' ...
 'UpA(#3#"Uplifts unfiltered")' ...
 'SubPto(#3#pantograph displacements")'];
R1.Post.SetFcn='';
% Now display tab with functional button
gf=cinguj('tabbedpanefig','demo_checkList');
R1=cinguj('ObjEditJ',R1,gf);
ua=struct('name','demo_checkList','ParentPanel',gf,'table',...
 {sdt_dialogs('uatable-end0','info','Post',R1)});
cinguj('tabbedpaneadd',gf,ua,ua.name,[]);
```

Element reference

bar1 386
beam1, beam1t 387
celas,cbush 390
dktp 394
fsc 396
hexa8, penta6, tetra4, and other 3D volumes . . . 399
integrules 400
mass1,mass2 409
m_elastic 410
m_heat 414
m_hyper 416
m_piezo 418
p_beam 420
p_heat 424
p_shell 428
p_solid 433
p_spring 436
p_super 438
p_piezo 440
quad4, quadb, mitc4 444
q4p, q8p, t3p, t6p and other 2D volumes . . . 447
rigid 448
tria3, tria6 451

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

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

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

Supported problem formulations are listed in section 6.1 , in particular one considers 2D and 3D elasticity, acoustics, hyperelasticity, fluid/structure coupling, piezo-electric volumes, ...

Other elements, non generic elements, are listed below

3-D plate/shell Elements	
dktp	3-node 9-DOF discrete Kirchoff plate
mitc4	4-node 20-DOF shell
quadb	quadrilateral 4-node 20/24-DOF plate/shell
quad9	(display only)
quadb	quadrilateral 8-node 40/48-DOF plate/shell
tria3	3-node 15/18-DOF thin plate/shell element
tria6	6-node 36DOF thin plate/shell element

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

Utility elements	
fe_super	element function for general superelement support
integrules	FEM integration rule support
fsc	fluid/structure coupling capabilities

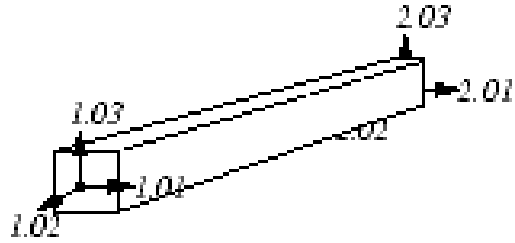
bar1

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 7.16).

```
[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

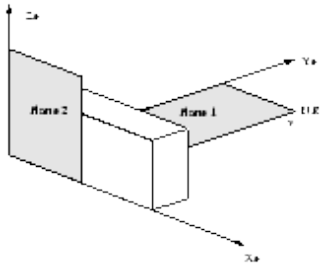
beam1, beam1t

Purpose

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

Description

`beam1`



In a model description matrix, *element property rows* for `beam1` elements follow the format

```
[n1 n2 MatID ProID nR 0 0 EltID p1 p2 x1 y1 z1 x2 y2 z2]
```

where

`n1,n2`	node numbers of the nodes connected
`MatID`	material property identification number
`ProID`	element section property identification number
`nr 0 0`	number of node not in the beam direction defining bending plane 1 in this case $\{v\}$ is the vector going from `n1` to `nr`. If `nr` is undefined it is assumed to be located at position [1.5 1.5 1.5].
`vx vy vz`	alternate method for defining the bending plane 1 by giving the components of a vector in the plane but not collinear to the beam axis. If `vy` and `vz` are zero, `vx` **must be negative or not an integer**. `MAP=beam1t('map',model)` returns a normal vector MAP giving the vector used for bending plane 1. This can be used to check your model.
`p1,p2`	pin flags. These give a list of DOFs to be released (condensed before assembly). For example, 456 will release all rotation degrees of freedom. Note that the DOFS are defined in the local element coordinate system.
`x1,...`	optional components in global coordinate system of offset vector at node 1 (default is no offset)
`x2,...`	optional components of offset vector at node 2

Isotropic elastic materials are the only supported (see m_elastic). p_beam describes the section property format and associated formulations.

Failure to define orientations is a typical error with beam models. In the following example, the definition of bending plane 1 using a vector is illustrated.

```
cf=feplot(femesh('test2bay'));
% Map is in very variable direction due to undefined nr
% This is only ok for sections invariant by rotation
beam1t('map',cf.mdl);fecom('view3');

% Now define generator for bending plane 1
i1=feutil('findelt eltname beam1',cf.mdl); % element row index
cf.mdl.Elt(i1,5:7)=ones(size(i1))*[-.1 .9 0]; % vx vy vz
beam1t('map',cf.mdl);fecom('view2');
```

beam1 adds secondary inertia effects which may be problematic for extremely short beams and beam1t may then be more suitable.

beam1t

For the bending part, this element solves

$$\rho A(\ddot{w} - \Omega^2 w) + \frac{\partial^2}{\partial x^2}\left(EI_y \frac{\partial^2 w}{\partial x^2}\right) - \frac{\partial}{\partial x}\left(T\frac{\partial w}{\partial x}\right) - f = 0 \tag{9.1}$$

with boundary conditions in transverse displacement

$$w = \text{given or} \quad F = T\frac{\partial w}{\partial x} - EI_y \frac{\partial^3 w}{\partial x^3} \tag{9.2}$$

and rotation

$$\frac{\partial w}{\partial x} = \text{ given or } \quad M = EI_y \frac{\partial^2 w}{\partial x^2} \tag{9.3}$$

This element has an internal state stored in a InfoAtNode structure where each column of Case.GroupInfo{7}.data gives the local basis, element length and tension [bas(:);L;ten]. Initial tension can be defined using a .MAP field in the element property.

This is a simple example showing 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('SetGroup 1 name beam1t',model);
d1=fe_eig(model,[5 10]);
model=feutil('setpro 112',model,'MAP', ...
```

```
  struct('dir',{{'1.5e6'}},'lab',{{'ten'}}));
d2=fe_eig(model,[5 10]);

figure(1);plot([d2.data./d1.data-1]);
xlabel('Mode index');ylabel('Frequency shift');
```

Strains in a non-linear Bernoulli Euler section are given by

$$\epsilon_{11} = \left(\frac{\partial u}{\partial x} + \frac{1}{2} \left(\frac{\partial {w_0}^2}{\partial x} \right) \right) - z \frac{\partial^2 w_0}{\partial x^2} \tag{9.4}$$

See also

p_beam, m_elastic, fe_mk, feplot

celas,cbush

Purpose

element function for scalar springs and penalized rigid links

Description

celas

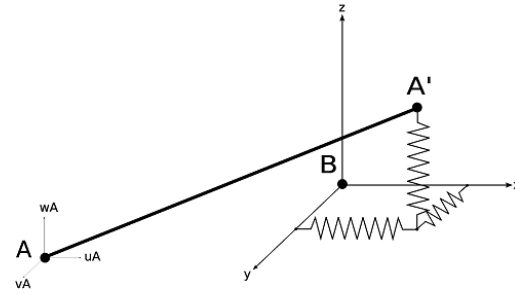
In an model description matrix a group of celas elements starts with a header row [Inf abs('celas') 0 ...] followed by element property rows following the format

[n1 n2 DofID1 DofID2 ProID EltID Kv Mv Cv Bv]

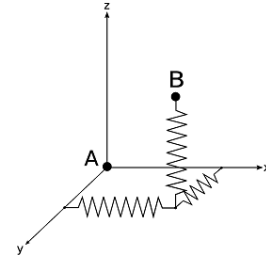
with

`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 of the other extremity slave node. Motion between the slave node and the second node is then penalized. `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 the penalization of node translations `2` 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. Use of negative `DofID1` will only activate additional DOF if explicitly given.



`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`.

`Bv` Optional stiffness hysteretic damping value : stiffness given by $K_v + iB_v$ (rather than $Kv(1 + i\eta)$ when using `p_spring`).

`p_spring` properties for `celas` elements take the form `[ProID type KvDefault m c eta S]`

By default a `celas` element will activate all 6 mechanical DOF in the model. If the `celas` element is not linked to other elements using these DOF (*e.g.* 3D elements do not use DOF 4-6), there will be a risk of null stiffness occurrence at assembly. To alleviate this problem use negative `DofID1` that will only activate additional DOF in the specified list. One can also fix the spurious DOF as a boundary condition.

Below is the example of a 2D beam on elastic supports.

```
model=femesh('Testbeam1 divide 10');
model=fe_case(model,'FixDof','2D',[.01;.02;.04]);
model.Elt(end+1,1:6)=[Inf abs('celas')]; % spring supports
model.Elt(end+[1:2],1:7)=[1 0 -13 0   0 0 1e5;2 0 -13 0   0 0 1e5];
def=fe_eig(model,[5 10 0]); feplot(model,def);
```

When using local displacement bases (non zero DID), the stiffness is defined in the local basis and transformed to global coordinates.

cbush

The element property row is defined by

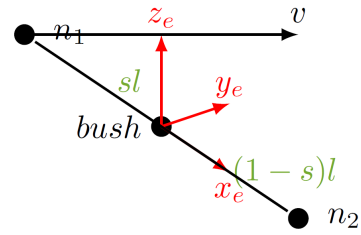
```
[n1 n2 MatId  ProId EltId x1  x2 x3 EDID  S OCID S1 S2 S3]
[n1 n2 MatId  ProId EltId NodeIdRef 0 0 EDID  S OCID S1 S2 S3]
```

The orientation of the spring (basis x_e, y_e, z_e) can be specified by

- `EDID>0` specifies a coordinate system for element orientation. This behaviour is pre-emptive.
- For **coincident** $n1, n2$, orientation vector given as `x1,x2,x3` can be used to specify x_e (this differs from figure and is not compatible with NASTRAN). To specify y_e for coincident nodes, you must use classically defined `EDID`, otherwise set the `EDID` to `-1`.
- For distinct `n1,n2`, default default orientations are in the global framework. Local directions require a non-null `EDID` to be activated. If one wants to use local directions defined in the element entry, on can use EDID=-1. Then, element x_e is along $n_2 - n_1$, other directions are defined as follows
  - giving orientation vector v as `x1,x2,x3` specifies y_e in the plane given by x_e and v. Note `x1` should not be an integer if `x2` and `x3` are zero to distinguish from the `NodeIdRef` case.
  - `NodeIdRef,0,0` specifies the use of a node number to create $v = n_{ref} - n_1$.

The spring/damper is located at a position interpolated between `n1` and `n2` using `S`, such that $x_i = Sn_1 + (1-S)n_2$. The midpoint is used by default, that-is-to-say `S` is taken at 0.5 if left to zero. To use other locations, specify a non-zero `OCID` and an offset `S1,S2,S3`.

It is possible to set n2 to 0 to define a grounded cbush.



See also

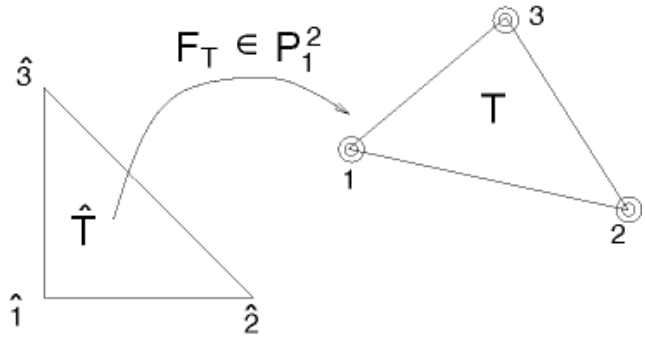
p_spring, rigid

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 :

$$W(x,y)\ ,\ \frac{\partial\, W}{\partial x}\ ,\ \frac{\partial\, W}{\partial y}\ ,\ \theta_x\ ,\ \theta_y\ \ (deflection\ W\ \ and\ \ rotations\ \ \theta) \tag{9.5}$$

- 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.

$$W(x, y) \ , \ \theta_x = (\frac{\partial \, W}{\partial x}) \ , \ \theta_y = (\frac{\partial \, W}{\partial y}) \tag{9.6}$$

The coordinates of the reference element's vertices are $\hat{S_1}(0., 0.)$, $\hat{S_2}(1., 0.)$ and $\hat{S_3}(0., 1.)$.

Surfaces are integrated using a 3 point rule $\omega_k = \frac{1}{3}$ and b_k mid side node.

See also

fe_mat, m_elastic, p_shell, fe_mk, feplot

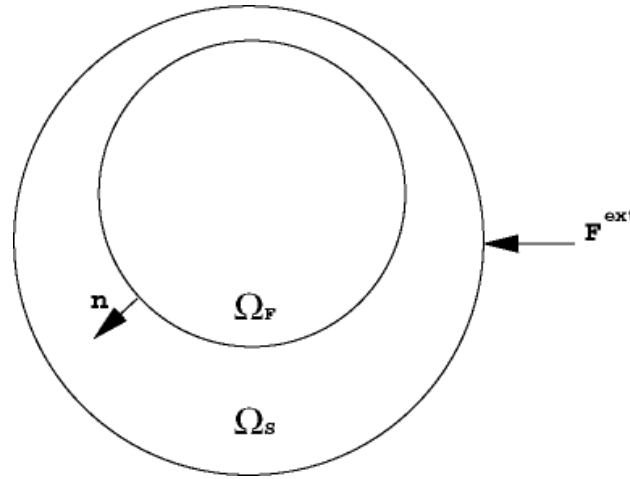
fsc

Purpose

Fluid structure/coupling with non-linear follower pressure support.

Description

Elasto-acoustic coupling is used to model structures containing a compressible, non-weighing fluid, with or without a free surface.



The FE formulation for this type of problem can be written as [44]

$$s^2 \begin{bmatrix} M & 0 \\ C^T & K_p \end{bmatrix} \begin{Bmatrix} q \\ p \end{Bmatrix} + \begin{bmatrix} K(s) & -C \\ 0 & F \end{bmatrix} \begin{Bmatrix} q \\ p \end{Bmatrix} = \begin{Bmatrix} F^{ext} \\ 0 \end{Bmatrix} \tag{9.7}$$

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

$$\begin{array}{c} \int_{\Omega_S} \sigma_{ij}(u)\epsilon_{ij}(\delta u)dx \Rightarrow \delta q^T K q \\ \int_{\Omega_S} \rho_S u.\delta u dx \Rightarrow \delta q^T M q \\ \frac{1}{\rho_F}\int_{\Omega_F} \nabla p \nabla \delta p dx \Rightarrow \delta p^T F p \\ \frac{1}{\rho_F c^2}\int_{\Omega_F} p\delta p dx \Rightarrow \delta p^T K_p p \\ \int_{\Sigma} p\delta u.n dx \Rightarrow \delta q^T C p \end{array} \tag{9.8}$$

To assemble fluid/structure coupling matrix you should declare a set of surface elements (any topology) with property `p_solid('dbval 1 fsc')`. The C matrix (solid forces induced by pressure field) is assembled with the stiffness (matrix type 1), while the C^T matrix (fluid pressure due to normal velocity of solid) is assembled with the mass (matrix type 2).

Some formulations, consider a surface impedance proportional to the pressure. This matrix can be computed by defining a group of surface elements with an acoustic material (see `m_elastic 2`) and a standard surface integration rule (`p_solid('dbval 1 d2 -3')`). This results in a mass given by

$$\delta p^T K_p p = \frac{1}{\rho_F c^2} \int_{\Omega_F} \delta p p dx \tag{9.9}$$

Follower force

One uses the identity

$$n\, dS = \frac{\partial \underline{x}}{\partial r} \wedge \frac{\partial \underline{x}}{\partial s}\, dr ds, \tag{9.10}$$

where (r, s) designate local coordinates of the face (assumed such that the normal is outgoing). Work of the pressure is thus:

$$\delta W_p = -\int_{r,s} \Pi\, (\frac{\partial \underline{x}}{\partial r} \wedge \frac{\partial \underline{x}}{\partial s}) \cdot \delta \underline{v}\, dr ds. \tag{9.11}$$

On thus must add the non-linear stiffness term:

$$-d\delta W_p = \int_{r,s} \Pi\, (\frac{\partial d\underline{u}}{\partial r} \wedge \frac{\partial \underline{x}}{\partial s} + \frac{\partial \underline{x}}{\partial r} \wedge \frac{\partial d\underline{u}}{\partial s}) \cdot \delta \underline{v}\, dr ds. \tag{9.12}$$

Using $\frac{\partial \underline{x}}{\partial r} = \{x_{1,r}\ x_{2,r}\ x_{3,r}\}^T$ (idem for s), and also

$$[Axr] = \begin{pmatrix} 0 & -x_{,r3} & x_{,r2} \\ x_{,r3} & 0 & -x_{,r1} \\ -x_{,r2} & x_{,r1} & 0 \end{pmatrix}, \quad [Axs] = \begin{pmatrix} 0 & -x_{,s3} & x_{,s2} \\ x_{,s3} & 0 & -x_{,s1} \\ -x_{,s2} & x_{,s1} & 0 \end{pmatrix}, \tag{9.13}$$

this results in

$$\begin{aligned} &(\tfrac{\partial d\underline{x}}{\partial r} \wedge \tfrac{\partial \underline{x}}{\partial s} + \tfrac{\partial \underline{x}}{\partial r} \wedge \tfrac{\partial d\underline{x}}{\partial s}) \cdot \delta \underline{v} = \\ &\quad \{\delta q_{ik}\}^T \{N_k\} (Axr_{ij} \{N_{l,s}\}^T - Axs_{ij} \{N_{l,r}\}^T) \{dq_j\}. \end{aligned} \tag{9.14}$$

Tests : `fsc3 testsimple` and `fsc3 test`.

In the RivlinCube test , the pressure on each free face is given by

$$\begin{array}{lllll} \Pi_1 = -\frac{1+\lambda_1}{(1+\lambda_2)(1+\lambda_3)}\Sigma_{11} & on & face & (x_1 = l_1) \\ \Pi_2 = -\frac{1+\lambda_2}{(1+\lambda_1)(1+\lambda_3)}\Sigma_{22} & on & face & (x_2 = l_2) \\ \Pi_3 = -\frac{1+\lambda_3}{(1+\lambda_1)(1+\lambda_2)}\Sigma_{33} & on & face & (x_3 = l_3). \end{array} \tag{9.15}$$

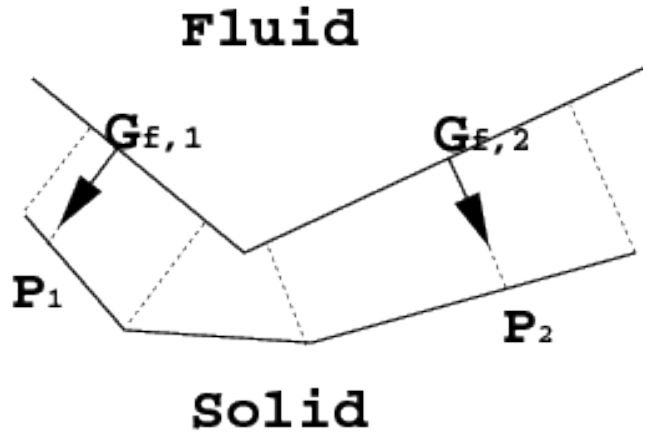
Non-conform

SDT supports non conforming element for fluid/structure coupling 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 center 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, penta6, tetra4, and other 3D volumes

Purpose

Topology holders for 3D volume elements.

Description

The hexa8 hexa20 hexa27, penta6 penta15 tetra4 and tetra10 elements are standard topology reference for 3D volume FEM problems.

In a model description matrix, **element property rows** for hexa8 and hexa20 elements follow the standard format with no element property used. The generic format for an element containing i nodes is [n1 ... ni MatID ProId EltId]. For example, the hexa8 format is [n1 n2 n3 n4 n5 n6 n7 n8 MatID ProId EltId].

These elements only define topologies, the nature of the problem to be solved should be specified using a property entry, see section 6.1 for supported problems and p_solid, p_heat, ... for formats.

Integration rules for various topologies are described under integrules. Vertex coordinates of the reference element can be found using an integrules command containing the name of the element such as r1=integrules('q4p');r1.xi.

Backward compatibility note : if no element property entry is defined, or with a p_solid entry with the integration rule set to zero, the element defaults to the historical 3D mechanic elements described in section 7.19.2 .

See also

fe_mat, m_elastic, fe_mk, feplot

.

integrules

Purpose

Command function for FEM integration rule support.

Description

This function groups integration rule manipulation utilities used by various elements. In terms of notations, a field u is interpolated within an element by shapes functions N_i and values of the field at nodes u_i

$$u(x,y,z) = \sum_i N_i(r,s,t)u_i \tag{9.16}$$

The relation between physical coordinates x, y, z and element coordinates r, s, t is itself described by a mapping associated with shape functions. When computing an integral, one selects a number of Gauss points r_g, s_g, t_g and associated weights w_g leading to an approximation of the integral as

$$\int_V f(x,y,z)dV \approx \sum_g f(r_g,s_g,t_g)Jw_g \tag{9.17}$$

where J is the determinant of the Jacobian of the transform from reference to physical coordinates. The field `.wjdet` is used to denote the local value of the product Jw_g. The following calls generate the reference `EltConst` data structure, see section 7.15.4 .

Gauss

This command supports the definition of Gauss points and associated weights. It is called with `integrules('Gauss Topology',RuleNumber)`. Supported topologies are `1d` (line), `q2d` (2D quadrangle), `t2d` (2D triangle), `t3d` (3D tetrahedron), `p3d` (3D prism), `h3d` (3D hexahedron).
`integrules('Gauss q2d')` will list available 2D quadrangle rules.

- `Integ -3` is always the default rule for the order of the element.
- `-2` a rule at nodes.
- `-1` the rule at center.

```
[ -3]    [ 0x1 double]    'element dependent default'
[ -2]    [ 0x1 double]    'node'
[ -1]    [ 1x4 double]    'center'
[102]    [ 4x4 double]    'gefdyn 2x2'
```

```
[  2]    [ 4x4 double]    'standard 2x2'
[109]    [ 9x4 double]    'Q4WT'
[103]    [ 9x4 double]    'gefdyn 3x3'
[104]    [16x4 double]    'gefdyn 4x4'
[  9]    [ 9x4 double]    '9 point'
[  3]    [ 9x4 double]    'standard 3x3'
[  2]    [ 4x4 double]    'standard 2x2'
[ 13]    [13x4 double]    '2x2 and 3x3'
```

bar1,beam1,beam3

For integration rule selection, these elements use the 1D rules which list you can find using `integrules('Gauss1d')`.

Geometric orientation convention for segment is • (1) → (2)

One can show the edge using *elt_name* `edge` (e.g. `beaml edge`).

t3p,t6p

Vertex coordinates of the reference element can be found using `r1=integrules('tria3');r1.xi`.

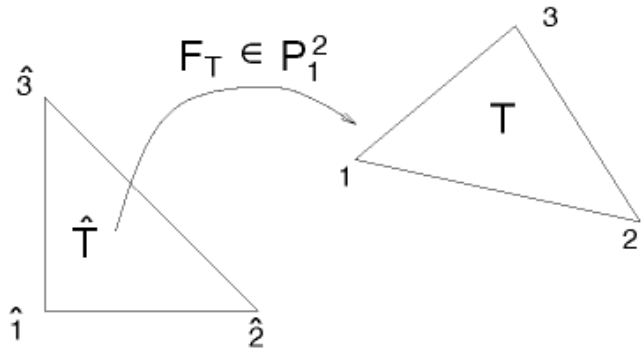


Figure 9.1: `t3p` reference element.

Vertex coordinates of the reference element can be found using `r1=integrules('tria6');r1.xi`.

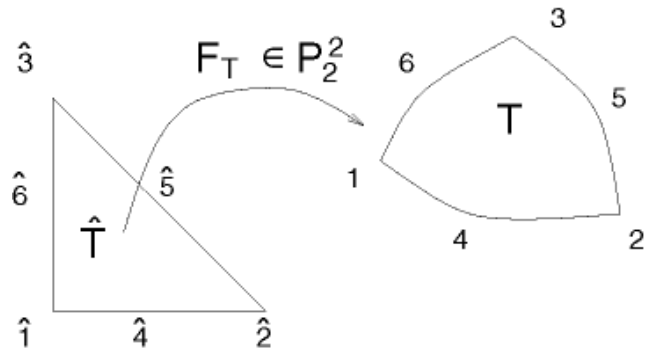


Figure 9.2: t6p reference element.

For integration rule selection, these elements use the 2D triangle rules which list you can find using integrules('Gausst2d').

Geometric orientation convention for triangle is to number anti-clockwise in the two-dimensional case (in the three-dimensional case, there is no orientation).
• edge [1]: (1) → (2) (nodes 4, 5,... if there are supplementary nodes) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (1) (...)

One can show the edges or faces using *elt_name* edge or *elt_name* face (e.g. t3p edge).

q4p, q5p, q8p

Vertex coordinates of the reference element can be found using r1=integrules('quad4');r1.xi.

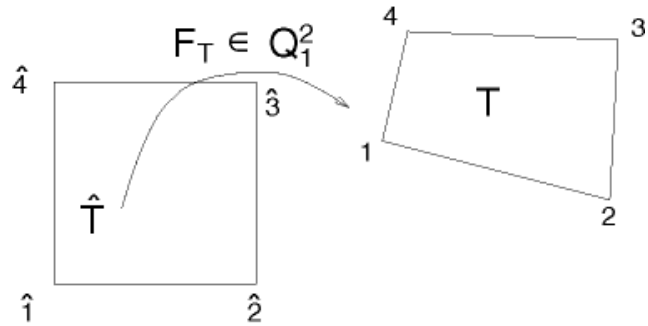


Figure 9.3: q4p reference element.

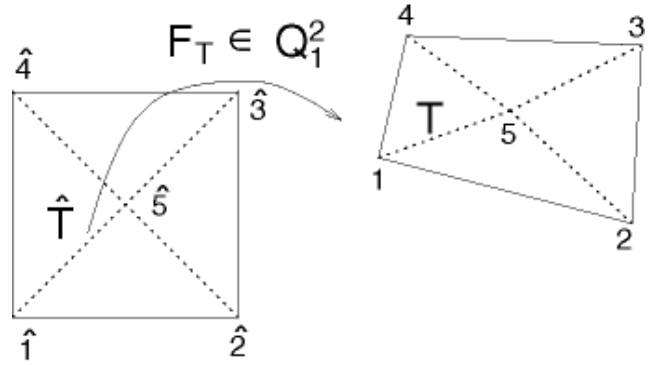


Figure 9.4: q5p reference element.

Vertex coordinates of the reference element can be found using the `r1=integrules('quadb');r1.xi`.

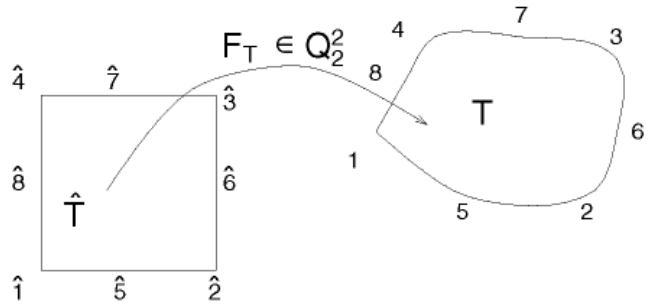


Figure 9.5: q8p reference element.

For integration rule selection, these elements use the 2D quadrangle rules which list you can find using `integrules('Gaussq2d')`.

Geometric orientation convention for quadrilateral is to number anti-clockwise (same remark as for the triangle)
• edge [1]: (1) → (2) (nodes 5, 6, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (4) • edge [4]: (4) → (1)

One can show the edges or faces using *elt_name* `edge` or *elt_name* `face` (e.g. `q4p edge`).

`tetra4,tetra10`

3D tetrahedron geometries with linear and quadratic shape functions. Vertex coordinates of the reference element can be found using `r1=integrules('tetra4');r1.xi` (or command `'tetra10'`).

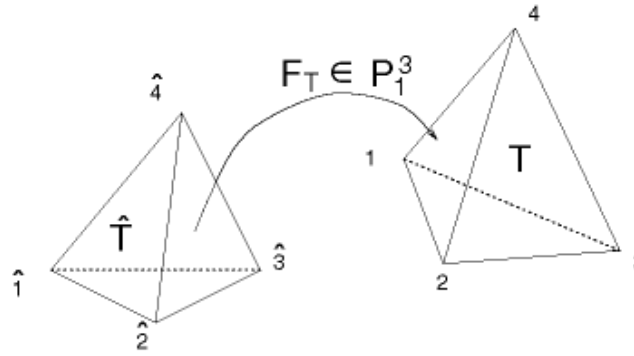


Figure 9.6: `tetra4` reference element.

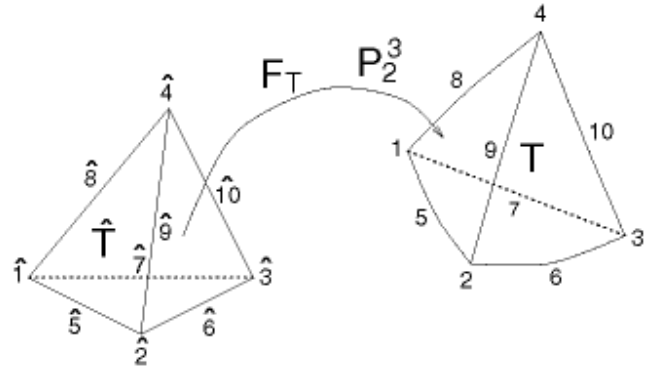


Figure 9.7: `tetra10` reference element.

For integration rule selection, these elements use the 3D pentahedron rules which list you can find using `integrules('Gausst3d')`.

Geometric orientation convention for tetrahedron is to have trihedral $(\vec{12}, \vec{13}, \vec{14})$ direct ($\vec{ij}$ designates the vector from point i to point j).
• edge [1]: (1) → (2) (nodes 5, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (1)
• edge [4]: (1) → (4) • edge [5]: (2) → (4) • edge [6]: (3) → (4) (nodes ..., p)

All faces, seen from the exterior, are described anti-clockwise:
• face [1]: (1) (3) (2) (nodes p+1, ...) • face [2]: (1) (4) (3) (...)
• face [3]: (1) (2) (4) • face [4]: (2) (3) (4)

One can show the edges or faces using *elt_name* `edge` or *elt_name* `face` (e.g. `tetra10 face`).

`penta6, penta15`

3D prism geometries with linear and quadratic shape functions. Vertex coordinates of the reference element can be found using `r1=integrules('penta6');r1.xi` (or command `'penta15'`).

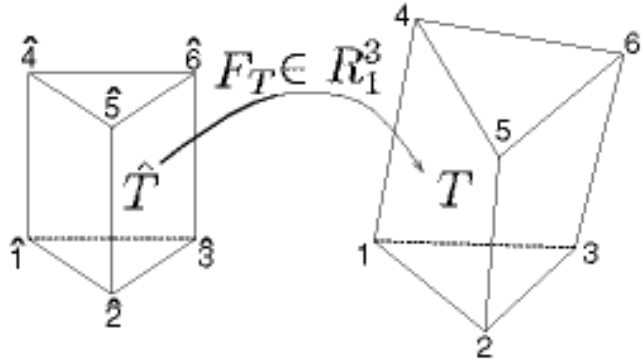


Figure 9.8: `penta6` reference element.

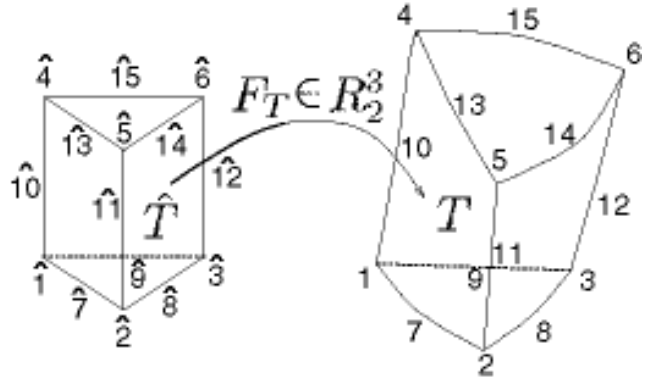


Figure 9.9: `penta15` reference element.

For integration rule selection, these elements use the 3D pentahedron rules which list you can find using `integrules('Gaussp3d')`.

Geometric orientation convention for pentahedron is to have trihedral $(\vec{12}, \vec{13}, \vec{14})$ direct
• edge [1]: (1) → (2) (nodes 7, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (1)
• edge [4]: (1) → (4) • edge [5]: (2) → (5) • edge [6]: (3) → (6)
• edge [7]: (4) → (5) • edge [8]: (5) → (6) • edge [9]: (6) → (4) (nodes ..., p)

All faces, seen from the exterior, are described anti-clockwise.
• face [1] : (1) (3) (2) (nodes p+1, ...) • face [2] : (1) (4) (6) (3) • face [3] : (1) (2) (5) (4)
• face [4] : (4) (5) (6) • face [5] : (2) (3) (5) (6)

One can show the edges or faces using *elt_name* edge or *elt_name* face (e.g. penta15 face).

hexa8, hexa20, hexa21, hexa27

3D brick geometries, using linear hexa8, and quadratic shape functions. Vertex coordinates of the reference element can be found using r1=integrules('hexa8');r1.xi (or command 'hexa20', 'hexa27').

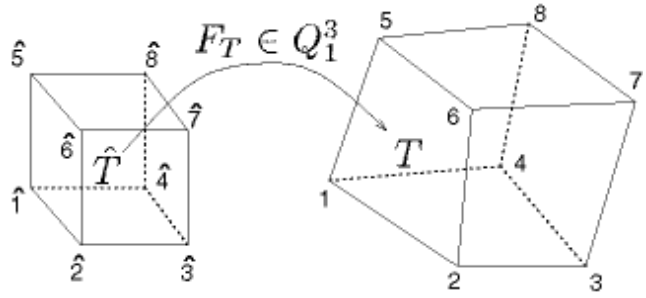


Figure 9.10: hexa8 reference topology.

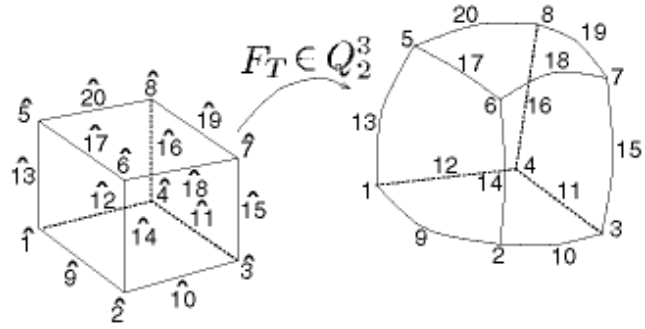


Figure 9.11: hexa20 reference topology.

For integration rule selection, these elements use the 3D hexahedron rules which list you can find using integrules('Gaussh3d').

Geometric orientation convention for hexahedron is to have trihedral $(\vec{12}, \vec{14}, \vec{15})$ direct
• edge [1]: (1) → (2) (nodes 9, ...) • edge [2]: (2) → (3) (...) • edge [3]: (3) → (4)

• edge [4]: (4) → (1) • edge [5]: (1) → (5) • edge [6]: (2) → (6)
• edge [7]: (3) → (7) • edge [8]: (4) → (8) • edge [9]: (5) → (6)
• edge [10]: (6) → (7) • edge [11]: (7) → (8) • edge [12]: (8) → (5) (nodes ..., p)

All faces, seen from the exterior, are described anti-clockwise.
• face [1] : (1) (4) (3) (2) (nodes p+1, ...) • face [2] : (1) (5) (8) (4)
• face [3] : (1) (2) (6) (5) • face [4] : (5) (6) (7) (8)
• face [5] : (2) (3) (7) (6) • face [6] : (3) (4) (8) (7)

One can show the edges or faces using *elt_name* edge or *elt_name* face (e.g. hexa8 face).

BuildNDN

The commands are extremely low level utilities to fill the .NDN field for a given set of nodes. The calling format is of_mk('BuildNDN',type,rule,nodeE) where type is an int32 that specifies the rule to be used : 2 for 2D, 3 for 3D, 31 for 3D with xyz sorting of NDN columns, 23 for surface in a 3D model, 13 for a 3D line. A negative value can be used to switch to the .m file implementation in integrules.

The 23 rule generates a transformation with the first axis along N, r, the second axis orthogonal in the plane tangent to N, r, N, s and the third axis locally normal to the element surface. If a local material orientation is provided in columns 5 to 7 of nodeE then the material x axis is defined by projection on the surface. One recalls that columns of nodeE are field based on the InfoAtNode field and the first three labels should be 'v1x','v1y','v1z'.

With the 32 rule if a local material orientation is provided in columns 5 to 7 for x and 8 to 10 for y the spatial derivatives of the shape functions are given in this local frame.

The rule structure is described earlier in this section and node has three columns that give the positions in the nodes of the current element. The rule.NDN and rule.jdet fields are modified. They must have the correct size before the call is made or severe crashes can be experienced.

If a rule.bas field is defined $(9 \times Nw)$, each column is filled to contain the local basis at the integration point for 23 and 13 types. If a rule.J field with $(4 \times Nw)$, each column is filled to contain the jacobian at the integration point for 23.

```
model=femesh('testhexa8');  nodeE=model.Node(:,5:7);
opt=integrules('hexa8',-1);
nodeE(:,5:10)=0; nodeE(:,7)=1;  nodeE(:,8)=1; % xe=z and ye=y
integrules('buildndn',32,opt,nodeE)

model=femesh('testquad4'); nodeE=model.Node(:,5:7);
opt=integrules('q4p',-1);opt.bas=zeros(9,opt.Nw);opt.J=zeros(4,opt.Nw);
nodeE(:,5:10)=0; nodeE(:,5:6)=1;  % xe= along [1,1,0]
```

```
integrules('buildndn',23,opt,nodeE)
```

See also

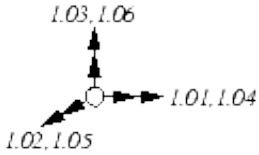
elem0

mass1,mass2

Purpose

Concentrated mass elements.

Description



`mass1` places a diagonal concentrated mass and inertia at one node.

In a model description matrix, **element property rows** for `mass1` elements follow the format

```
[NodeID mxx myy mzz ixx iyy izz EltID]
```

where the concentrated nodal mass associated to the DOFs `.01` to `.06` of the indicated node is given by

```
diag([mxx myy mzz ixx iyy izz])
```

Note `feutil GetDof` eliminates DOFs where the inertia is zero. You should thus use a small but non zero mass to force the use of all six DOFs.

For `mass2` elements, the **element property rows** follow the format

```
[n1 M I11 I21 I22 I31 I32 I33 EltID CID X1 X2 X3 MatId ProId]
```

which, for no offset, corresponds to matrices given by

$$\begin{bmatrix} M & & \text{symmetric} & & & \\ & M & & & & \\ & & M & & & \\ & & & I_{11} & & \\ & & & -I_{21} & I_{22} & \\ & & & -I_{31} & -I_{32} & I_{33} \end{bmatrix} = \begin{bmatrix} \int \rho dV & & \text{symmetric} & & & \\ & M & & & & \\ & & M & & & \\ & & & \int \rho(x^2+y^2)dV & & \\ & & & -I_{21} & I_{22} & \\ & & & -I_{31} & -I_{32} & I_{33} \end{bmatrix} \tag{9.18}$$

Note that local coordinates `CID` are not currently supported by `mass2` elements.

See also

`femesh`, `feplot`

m_elastic

Purpose

Material function for elastic solids and fluids.

Syntax

```
mat= m_elastic('default')
mat= m_elastic('database name')
mat= m_elastic('database -therm name')
pl = m_elastic('dbval MatId name');
pl = m_elastic('dbval -unit TM MatId name');
pl = m_elastic('dbval -punit TM MatId name');
pl = m_elastic('dbval -therm MatId name');
```

Description

This help starts by describing the main commands of `m_elastic` : `Database` and `Dbval`. Materials formats supported by `m_elastic` are then 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', ...
  'dbval 200 lamina .27 3e9 .4 1200 0  790e9 .3 1780 0');
pl=fe_mat('convert SITM',pl);
pl=m_elastic(pl,'dbval -unit TM 102 steel')
```

Command option `-unit` asks the output to be converted in the desired unit system. Command option `-punit` tells the function that the provided data is in a desired unit system (and generates the corresponding type). Command option `-therm` asks to keep thermal data (linear expansion coefficients and reference temperature) if existing.

You can generate orthotropic shell properties using the `Dbval 100 lamina VolFrac Ef nu_f rho_f G_f E_m nu_m Rho_m G_m` command which gives fiber and matrix characteristics as illustrated above (the volume fraction is that of fiber).

The default material is steel.

To orient fully anisotropic materials, you can use the following command

```
% Behavior of a material grain assumed orthotropic
C11=168.4e9; C12=121.4e9; C44=75.4e9; % GPa
C=[C11 C12 C12 0 0 0;C12 C11 C12 0 0 0;C12 C12 C11 0 0 0;
  0 0 0 C44 0 0;    0 0 0 0 C44 0;    0 0 0 0 0 C44];

pl=[m_elastic('formulaPlAniso 1',C,basis('bunge',[5.175 1.3071 4.2012]));
    m_elastic('formulaPlAniso 2',C,basis('bunge',[2.9208 1.7377 1.3921]))];
```

Subtypes `m_elastic` supports the following material subtypes

`1 : standard isotropic`

Standard isotropic materials, see section 6.1.1 and section 6.1.2 , are described by a row of the form

```
[MatID   typ   E nu rho G Eta Alpha T0]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',1)` command, E (Young's modulus), ν (Poisson's ratio), ρ (density), G (shear modulus, set to $G = E/2(1+\nu)$ if equal to zero). η loss factor for hysteretic damping modeling. α thermal expansion coefficient. T_0 reference temperature. $G = E/2(1+\nu)$

By default E and G are interdependent through $G = E/2(1+\nu)$. One can thus define either E and G to use this property. If E or G are set to zero they are replaced on the fly by their theoretical expression. Beware that modifying only E or G, either using `feutilSetMat` or by hand, will not apply modification to the other coefficient. In case where both coefficients are defined, in thus has to modify both values accordingly.

`2 : acoustic fluid`

Acoustic fluid , see section 6.1.3 ,are described by a row of the form

```
[MatId typ rho C eta R]
```

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 $K = \rho C^2$.

For walls with an impedance (see p_solid 3 form 8), the real part of the impedance, which corresponds to a viscous damping on the wall is given by $Z = \rho CR$. If an imaginary part is to be present, one will use $Z = \rho CR(1 + i\eta)$. In an acoustic tube the absorbtion factor is given by $\alpha = \frac{4R}{((R+1)^2+(R\eta)^2)}$.

3 : 3-D anisotropic solid

3-D Anisotropic solid, see section 6.1.1 , are described by a row of the form

```
[MatId typ Gij rho eta A1 A2 A3 A4 A5 A6 T0]
```

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]
```

Note that shear is ordered g_yz, g_zx, g_xy which may not be the convention of other software.

SDT supports material handling through

- material bases defined for each element xx
- orientation maps used for material handling are described in section 7.13 . It is then expected that the six components v1x,v1y,v1z,v2x,v2y,v2z are stored sequentially in the interpolation table.It is then usual to store the MAP in the stack entry info,EltOrient.

4 : 2-D anisotropic solid

2-D Anisotropic solid, see section 6.1.2 , are described by a row of the form

```
[MatId typ E11 E12 E22 E13 E23 E33 rho eta a1 a2 a3 T0]
```

with typ an identifier generated with the fe_mat('m_elastic','SI',4) command, *rho* (density), *eta* (loss factor) and *Eij* elastic constants and *ai* anisotropic thermal expansion coefficients.

5 : shell orthotropic material

shell orthotropic material, see section 6.1.4 corresponding to NASTRAN MAT8, are described by a row of the form

```
[MatId typ E1 E2 nu12 G12 G1z G2z Rho A1 A2 T0 Xt Xc Yt Yc S Eta ...
  F12 STRN]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',5)` command, *rho* (density), ... See `m_elastic Dbvallamina` for building.

6 : Orthotropic material

3-D orthotropic material, see section 6.1.1 , are described by a set of engineering constants, in a row of the form

```
[MatId typ E1 E2 E3 Nu23 Nu31 Nu12 G23 G31 G12 rho a1 a2 a3 T0 eta]
```

with `typ` an identifier generated with the `fe_mat('m_elastic','SI',6)` command, *Ei* (Young modulus in each direction), νij (Poisson ratio), *Gij* (shear modulus), *rho* (density), *ai* (anisotropic thermal expansion coefficient), T_0 (reference temperature), and *eta* (loss factor). Care must be taken when using these conventions, in particular, it must be noticed that

$$\nu_{ji} = \frac{E_j}{E_i}\nu_{ij} \tag{9.19}$$

See also

Section 4.5.1, section 7.3 , `fe_mat`, `p_shell`

m_heat

Purpose

Material function for heat problem elements.

Syntax

```
mat= m_heat('default')
mat= m_heat('database name')
pl = m_heat('dbval MatId name');
pl = m_heat('dbval -unit TM MatId name');
pl = m_heat('dbval -punit TM MatId name');
```

Description

This help starts by describing the main commands of m_heat : Database and Dbval. Materials formats supported by m_heat are then 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_heat('DataBase Steel') returns a the data structure describing steel.
m_heat('DBVal 100 Steel') only returns the property row.

```
% List of materials in data base
m_heat info
% examples of row building and conversion
pl=m_heat('DBVal 5 steel');
pl=m_heat(pl,...
  'dbval 101 aluminum', ...
  'dbval 200 steel');
pl=fe_mat('convert SITM',pl);
pl=m_heat(pl,'dbval -unit TM 102 steel')
```

Subtypes m_heat supports the following material subtype

1 : Heat equation material
[MatId fe_mat('m_heat','SI',2) k rho C Hf]

- k conductivity
- rho mass density
- C heat capacity
- Hf heat exchange coefficient

See also

Section 4.5.1, section 7.3 , fe_mat, p_heat

m_hyper

Purpose

Material function for hyperelastic solids.

Syntax

```
mat= m_hyper('default')
mat= m_hyper('database name')
pl = m_hyper('dbval MatId name');
pl = m_hyper('dbval -unit TM MatId name');
pl = m_hyper('dbval -punit TM MatId name');
```

Description

Function based on `m_elastic` function adapted for hyperelastic material. Only subtype 1 is currently used:

```
1 :  Nominal hyperelastic material
```

Nominal hyperelastic materials are described by a row of the form

```
[MatID   typ  rho Wtype C_1 C_2 K]
```

with `typ` an identifier generated with the `fe_mat('m_hyper','SI',1)` command, *rho* (density), *Wtype* (value for Energy choice), C_1, C_2, K (energy coefficients).
Possible values for *Wtype* are:

$$\begin{aligned} 0: &\quad W = C_1(J_1-3) + C_2(J_2-3) + K(J_3-1)^2 \\ 1: &\quad W = C_1(J_1-3) + C_2(J_2-3) + K(J_3-1) - (C_1+2C_2+K)\ln(J_3) \end{aligned}$$

Other energy functions can be added by editing the `hyper.c Enpassiv` function.

In RivlinCube test, m_hyper is called in this form:

```
model.pl=m_hyper('dbval 100 Ref'); % this is where the material is defined
```

the hyperelastic material called "Ref" is described in the database of `m_hyper.m` file:

```
out.pl=[MatId fe_mat('type','m_hyper','SI',1) 1e-06 0 .3 .2 .3];
out.name='Ref';
out.type='m_hyper';
out.unit='SI';
```

Here is an example to set your material property for a given structure model:

```
model.pl = [MatID fe_mat('m_hyper','SI',1) typ rho Wtype C_1 C_2 K];
model.Elt(2:end,length(feval(ElemF,'node')+1)) = MatID;
```

m_piezo

Purpose

Material function for piezoelectric solids

Syntax

```
mat= m_piezo('database name')
pl = m_piezo('dbval MatId  -elas 12 Name');
```

See section 6.1.5 for tutorial calls. Accepted commands are

[Database, Dbval] [-unit TY] [,MatiD]] Name

m_piezo contains a number of defaults obtained with the database and dbval commands which respectively return a structure or an element property row. You can select a particular entry of the database with using a name matching the database entries.

Piezoelectric materials are associated with two material identifiers, the main defines the piezoelectric properties and contains a reference ElasMatId to an elastic material used for the elastic properties of the material (see m_elastic for input formats).

```
m_piezo('info') % List of materials in data base
% database piezo and elastic properties
pl=m_piezo('dbval 3 -elas 12 Sample_ULB')
```

Theoretical details on piezoelectric materials are given in chapter 6.1.5. The m_piezo Const and BuildConstit commands support integration constant building for piezo electric volumes integrated in the standard volume elements. Element properties are given by p_solid entries, while materials formats are detailed here.

Patch

Supports the specification of a number of patches available on the market. The call uses an option structure with fields

- .name of the form ProId*val+patchName*. For example ProId1+SmartM.MFC-P1.2814.
- MatId value for the initial MatId.

m_piezo('patch') lists currently implemented geometries. In particular

- Noliac.Material.Geometry is used for circular patches by Noliac. Fields for the geometry are
  - OD outer diameter (mm).
  - TH Thickness (mm). To specify a milimiter fraction replace the . by and _. For example TH0_7 is used for TH=0.7 mm.
  - ID inner diameter (mm) (optional for piezo rings).
- SmartM.Material.Geometry is used for circular patches by Noliac. The geometry is coded assuming a rectangle in mm. Thus 2814 corresponds to an 28 x 14 mm active rectangle.

The piezoelectric constants can be declared using the following sub-types

1 : Simplified 3D piezoelectric properties

[ProId Type ElasMatId d31 d32 d33 eps1T eps2T eps3T EDType]

These simplified piezoelectric properties (1.4) can be used for PVDF, but also for PZT if shear mode actuation/sensing is not considered ($d_{24} = d_{15} = 0$). For EDType==0 on assumes d is given. For EDType==1, e is given. Note that the values of ε^T (permitivity at zero stress) should be given (and not ε^S).

2 : General 3D piezo

[ProId Type ElasMatId d_1:18 epsT_1:9]

d_1:18 are the 18 constants of the $[d]$ matrix (see section 6.1.5), and epsT_1:9 are the 9 constants of the $\left[\varepsilon^T\right]$ matrix. One reminds that strains are stored in order xx, yy, zz, yz, zx, yx.

3 : General 3D piezo, e matrix

[ProId Type ElasMatId e_1:18 epsT_1:9]

e_1:18 are the 18 constants of the $[d]$ matrix, and epsT_1:9 are the 9 constants of the $\left[\varepsilon^T\right]$ matrix in the constitutive law (see section 6.1.5).

See also

p_piezo.

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');
il = p_beam('dbval -punit TM ProId name');
il2= p_beam('ConvertTo1',il)
```

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*. See beam1 for orientation (the default reference node is 1.5, 1.5, 1.5 so that orientation MUST be defined for non-symmetric sections).
Type *r1 r2 ...*	other predefined sections of subtype 3 are listed using p_beam('info').

For example, you will obtain the section property row with ProId 100 associated with a circular cross section of $0.05m$ or a rectangular $0.05 \times 0.01m$ cross section using

```
% ProId 100, rectangle 0.05 m by 0.01 m
pro = p_beam('database 100 rectangle .05 .01')
% ProId 101 circle radius .05
il = p_beam(pro.il,'dbval 101 circle .05')
p_beam('info')
% ProId 103 tube external radius .05 internal .04
```

```
 il = p_beam(il,'dbval -unit SI 103 tube .05 .04')
 % Transform to subtype 1
 il2=p_beam('ConvertTo1',il)
 il(end+1,1:6)=[104 fe_mat('p_beam','SI',1) 0 0 0 1e-5];
 il = fe_mat('convert SITM',il);
% Generate a property in TM, providing data in SI
 il = p_beam(il,'dbval -unit TM 105 rectangle .05 .01')
% Generate a property in TM providing data in TM
  il = p_beam(il,'dbval -punit TM 105 rectangle 50 10')
```

Show3D,MAP ...

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 4.5.1 . For a programmers reference on formats used to describe element properties see section 7.4 .

```
1 :  standard
     [ProID   type   J I1 I2 A   k1 k2 lump NSM]
```

ProID	element property identification number.
type	identifier obtained with fe_mat('p_beam','SI',1).
J	torsional stiffness parameter (often different from polar moment of inertia I1+I2).
I1	moment of inertia for bending plane 1 defined by a third node nr or the vector vx vy vz (defined in the beam1 element). For a case with a beam along x and plane 1 the xy plane I1 is equal to $Iz = \int_S y^2 ds$.
I2	moment of inertia for bending plane 2 (containing the beam and orthogonal to plane 1.
A	section area.
k1	(optional) shear factor for motion in plane 1 (when not 0, a Timoshenko beam element is used). The effective area of shear is given by $k_1 A$.
k2	(optional) shear factor for direction 2.
lump	(optional) request for lumped mass model. 1 for inclusion of inertia terms. 2 for simple half mass at node.
NSM	(optional) non structural mass (density per unit length).

bar1 elements only use the section area. All other parameters are ignored.

beam1 elements use all parameters. Without correction factors (*k1 k2* not given or set to 0), the beam1 element is the standard Bernoulli-Euler 12 DOF element based on linear interpolations for traction and torsion and cubic interpolations for flexion (see Ref. [37] 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 [45]. No correction for rotational inertia of sections is used.

3 : Cross section database

This subtype can be used to refer to standard cross sections defined in database. It is particularly used by nasread when importing NASTRAN PBEAML properties.

```
[ProID   type   0  Section Dim(i) ... ]
```

ProID	element property identification number.
type	identifier obtained with fe_mat('p_beam','SI',3).
Section	identifier of the cross section obtained with comstr('*SectionName*',-32' where *SectionName* is a string defining the section (see below).
Dim1 ...	dimensions of the cross section.

Cross section, if existing, is compatible with NASTRAN PBEAML definition. Equivalent moment of inertia and tensional stiffness are computed at the centroid of the section. Currently available sections are listed with p_beam('info'). In particular one has ROD (1 dim), TUBE (2 dims), T (4 dims), T2 (4 dims), I (6 dims), BAR (2 dims), CHAN1 (4 dims), CHAN2 (4 dims).

For NSM and Lump support ConverTo1 is used during definition to obtain the equivalent subtype 1 entry.

See also

Section 4.5.1, section 7.4 , fe_mat

p_heat

Purpose

Formulation and material support for the heat equation.

Syntax

```
il = p_heat('default')
```

Description

This help starts by describing the main commands : p_heat Database and Dbval. Supported p_heat subtypes and their formats are then described. For theory see section 6.1.13 .

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.

p_heat database

```
il=p_heat('database');
```

Accepted commands for the database are

- d3 *Integ SubType* : Integ integration rule for 3D volumes (default -3).
- d2 *Integ SubType* : Integ integration rule for 2D volumes (default -3).

For fixed values, use p_heat('info').

Example of database property construction

```
il=p_heat([100 fe_mat('p_heat','SI',1) 0 -3 3],...
           'dbval 101 d3 -3 2');
```

Heat equation element properties

Element properties are described by the row of an element property matrix or a data structure with an .il field containing this row. Element property functions such as p_beam support graphical editing of properties and a database of standard properties.

1 : Volume element for heat diffusion (dimension DIM)
[ProId fe_mat('p_heat','SI',1) CoordM Integ DIM]

ProID	element property identification number
type	identifier obtained with fe_mat('p_beam','SI',1)
Integ	is rule number in integrules
DIM	is problem dimension 2 or 3 D

2 : Surface element for heat exchange (dimension DIM-1)
[ProId fe_mat('p_heat','SI',2) CoordM Integ DIM]

ProID	element property identification number
type	identifier obtained with fe_mat('p_beam','SI',2)
Integ	is rule number in integrules
DIM	is problem dimension 2 or 3 D

SetFace

This command can be used to define a surface exchange and optionally associated load. Surface exchange elements add a stiffness term to the stiffness matrix related to the exchange coefficient Hf defined in corresponding material property. One then should add a load corresponding to the exchange with the source temperature at T_0 through a convection coefficient Hf which is Hf.T_0. If not defined, the exchange is done with source at temperature equal to 0.

```
model=p_heat('SetFace',model,SelElt,pl,il);
```

- SelElt is a findelt command string to find faces that exchange heat (use 'SelFace' to select face of a given preselected element).
- pl is the identifier of existing material property (MatId), or a vector defining an m_heat property.
- il is the identifier of existing element property (ProId), or a vector defining an p_heat property.

Command option -load *T* can be used to defined associated load, for exchange with fluid at temperature *T*. Note that if you modify Hf in surface exchange material property you have to update the load.

Following example defines a simple cube that exchanges with thermal source at 55 deg on the bottom face.

```
model=femesh('TestHexa8'); % Build simple cube model
model.pl=m_heat('dbval 100 steel'); % define steel heat diffusion parameter
model.il=p_heat('dbval 111 d3 -3 1'); % volume heat diffusion (1)
model=p_heat('SetFace-load55',... % exchange at 55 deg
    model,...
    'SelFace & InNode{z==0}',... % on the bottom face
    100,... % keep same matid for exchange coef
    p_heat('dbval 1111 d3 -3 2')); % define 3d, integ-3, for surface exchange (2)
cf=feplot(model); fecom colordatapro
def=fe_simul('Static',model); % compute static thermal state
mean(def.def)
```

2Dvalidation

Consider a bi-dimensional annular thick domain Ω with radii $r_e = 1$ and $r_i = 0.5$. The data are specified on the internal circle Γ_i and on the external circle Γ_e. The solid is made of homogeneous isotropic material, and its conductivity tensor thus reduces to a constant k. The steady state temperature distribution is then given by

$$-k\Delta\theta(x,y) = f(x,y) \quad in \quad \Omega. \tag{9.20}$$

The solid is subject to the following boundary conditions

- $\Gamma_i\,(r = r_i)$: Neumann condition

$$\frac{\partial\theta}{\partial n}(x,y) = g(x,y) \tag{9.21}$$

- $\Gamma_e\,(r = r_e)$: Dirichlet condition

$$\theta(x,y) = \theta_{ext}(x,y) \tag{9.22}$$

In above expressions, f is an internal heat source, θ_{ext} an external temperature at $r = r_e$, and g a function. All the variables depend on the variable x and y.

The OpenFEM model for this example can be found in `ofdemos('AnnularHeat')`.
Numerical application : assuming $k = 1$, $f = 0$, $Hf = 1e^{-10}$, $\theta_{ext}(x,y) = \exp(x)\cos(y)$ and

$g(x,y) = -\frac{\exp(x)}{r_i}(\cos(y)x - \sin(y)x)$, the solution of the problem is given by $\theta(x,y) = \exp(x)\cos(y)$

See also

section 6.1.13 , section 4.5.1 , fe_mat

p_shell

Purpose

Element property function for shells and plates (flat shells)

Syntax

```
il = p_shell('default');
il = p_shell('database ProId name');
il = p_shell('dbval ProId name');
il = p_shell('dbval -unit TM ProId name');
il = p_shell('dbval -punit TM ProId name');
il = p_shell('SetDrill 0',il);
```

Description

This help starts by describing the main commands : p_shell Database and Dbval. Supported p_shell subtypes and their formats are then described.

Database,Dbval] ...

p_shell contains a number of defaults obtained with the database and dbval commands which respectively return a structure or an 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. By default the z values are counted from -thick/2, you can specify another value with a z0.

You can append a string option of the form -f *i* to select the appropriate shell formulation. For example, you will obtain the element property row with ProId 100 associated with a .1 thick Kirchhoff shell (with formulation 5) or the corresponding Mindlin plate use

```
il = p_shell('database 100 MindLin .1')
il = p_shell('dbval 100 kirchhoff .1 -f5')
il = p_shell('dbval 100 laminate z0=-2e-3 110 3e-3 30 110 3e-3 -30')
il = fe_mat('convert SITM',il);
il = p_shell(il,'dbval -unit TM 2 MindLin .1') % set in TM, provide data in SI
il = p_shell(il,'dbval -punit TM 2 MindLin 100') % set in TM, provide data in TM
```

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 4.5.1 . For a 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 O   h   k   MID2 RatI12_T3 MID3 NSM Z1 Z2 MID4]
```

type		identifier obtained with fe_mat('p_shell','SI',1).
f		0 use default of element. For other formulations the specific help for each element (quad4, tria3, ...), each formulation specifies integration rule.
d	-1	no drilling stiffness. The element DOFs are the standard translations and rotations at all nodes (DOFs .01 to .06). The drill DOF (rotation .06 for a plate in the *xy* plane) has no stiffness and is thus eliminated by fe_mk if it corresponds to a global DOF direction. The default is d=1 (d is set to 1 for a declared value of zero).
	d	arbitrary drilling stiffness with value proportional to d is added. This stiffness is often needed in shell problems but may lead to numerical conditioning problems if the stiffness value is very different from other physical stiffness values. Start with a value of 1. Use il=p_shell('SetDrill d',il) to set to d the drilling stiffness of all p_shell subtype 1 rows of the property matrix il.
h		plate thickness.
k	*k*	shear correction factor (default 5/6, default used if k is zero). This correction is not used for formulations based on triangles since tria3 is a thin plate element.
RatI12_T3		Ratio of bending moment of inertia to nominal T3/I12 (default 1).
NSM		Non structural mass per unit area.
MID2		material property for bending. Defauts to element MatId if equal to 0.
MID3		material property for transverse shear.
z1,z2		(unused) offset for fiber computations.
MID4		material property for membrane/bending coupling.

Shell strain is defined by the membrane, curvature and transverse shear

(display with `p_shell('ConstShell')`).

$$\begin{Bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ 2\epsilon_{xy} \\ \kappa_{xx} \\ \kappa_{yy} \\ 2\kappa_{xy} \\ \gamma_{xz} \\ \gamma_{yz} \end{Bmatrix} = \begin{bmatrix} N,x & 0 & 0 & 0 & 0 \\ 0 & N,y & 0 & 0 & 0 \\ N,y & N,x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & N,x \\ 0 & 0 & 0 & -N,y & 0 \\ 0 & 0 & 0 & -N,x & N,y \\ 0 & 0 & N,x & 0 & -N \\ 0 & 0 & N,y & N & 0 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \\ ru \\ rv \end{Bmatrix} \tag{9.23}$$

`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.
`Eta`	Hysteretic loss factor.
`LAM`	Laminate type.
`MatIdi`	`MatId` for ply *i*, see `m_elastic 1`, `m_elastic 5`, ...
`Ti`	Thickness of ply *i*.
`Thetai`	Orientation of ply *i*.
`SOUTi`	Stress output request for ply *i*.

Note that this subtype is based on the format used by NASTRAN for `PCOMP` and the formulation used for each topology is discussed in each element (see `quad4`, `tria3`). You can use the `DbvalLaminate` commands to generate standard entries.

$$\begin{Bmatrix} N \\ M \\ Q \end{Bmatrix} = \begin{bmatrix} A & B & 0 \\ B & D & 0 \\ 0 & 0 & F \end{bmatrix} \begin{Bmatrix} \epsilon \\ \kappa \\ \gamma \end{Bmatrix} \tag{9.24}$$

setTheta

When dealing with laminated plates, the classical approach uses a material orientation constant per element. OpenFEM also supports more advanced strategies with orientation defined at nodes but this is still poorly documented.

The material orientation is the reference for plies. Any angle defined in a laminate command is an additional rotation. In the example below, the element orientation is rotated 30 degrees, and the ply another 30. The fibers are thus oriented 60 degrees in the xy plane. Stresses are however given in the material orientation thus with a 30 degree rotation. Per ply output is not currently implemented.

The element-wise material angle is stored for each element. In column 7 for tria3, 8 for quad4, ... The setTheta command is a utility to ease the setting of these angles. By default, the orientation is done at element center. To use the mean orientation at nodes use command option -strategy 2.

```
model=ofdemos('composite');
model.il = p_shell('dbval 110 laminate 100 1 30'); % single ply

% Define material angle based on direction at element
MAP=feutil('getnormalElt MAP -dir1',model);
bas=basis('rotate',[],'rz=30;',1);
MAP.normal=MAP.normal*reshape(bas(7:15),3,3)';
model=p_shell('setTheta',model,MAP);

% Obtain a MAP of material orientations
MAP=feutil('getnormalElt MAP -dir1',model);
feplot(model);fecom('showmap',MAP)

% Set elementwise material angles using directions given at nodes.
% Here a global direction
MAP=struct('normal',ones(size(model.Node,1),1)*bas(7:9), ...
    'ID',model.Node(:,1),'opt',2);
model=p_shell('setTheta',model,MAP);

% Using an analytic expression to define components of
% material orientation vector at nodes
data=struct('sel','groupall','dir',{{'x-0','y+.01',0}},'DOF',[.01;.02;.03]);
model=p_shell('setTheta',model,data);
MAP=feutil('getnormalElt MAP -dir1',model);
feplot(model);fecom('showmap',MAP)
```

model=p_shell('setTheta',model,0) is used to reset the material orientation to zero.

Technically, shells use the `of_mk('BuildNDN')` rule 23 which generates a basis at each integration point. The first vector `v1x,v1y,v1z` is built in the direction of r lines and `v2x,v2y,v2z` is tangent to the surface and orthogonal to $v1$. When a `InfoAtNode` map provides `v1x,v1y,v1z`, this vector is projected (NEED TO VERIFY) onto the surface and $v2$ taken to be orthogonal.

See also

Section 4.5.1, section 7.4 , `fe_mat`

p_solid

Purpose

Element property function for volume elements.

Syntax

```
il=p_solid('database ProId Value')
il=p_solid('dbval ProId Value')
il=p_solid('dbval -unit TM ProId name');
il=p_solid('dbval -punit TM ProId name');
model=p_solid('default',model)
```

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,Default] ...

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 commands for the database are

- d3 *Integ* : *Integ* integration rule for quadratic 3D volumes. For information on rules available see integrules Gauss. Examples are d3 2 2x2x2 integration rule for linear volumes (hexa8 ...); d3 -3 default integration for all 3D elements, ...
- d2 *Integ* : *Integ* integration rule for quadratic 2D volumes. For example d2 2 2x2x2 integration rule for linear volumes (q4p ...). You can also use d2 1 0 2 for plane stress, and d2 2 0 2 for axisymmetry.
- fsc *Integ* : integration rule selection for fluid/structure coupling.

For fixed values, use p_solid('info').

For a tutorial on material/element property handling see section 4.5.1 . For a 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','dbval 102 d3 -3');
il=fe_mat('convert SITM',il);
il=p_solid(il,'dbval -unit TM 2 Reduced shear')
% Try a smart guess on default
model=femesh('TestHexa8');model.il=[];
model=p_solid('default',model)
```

1 : 3D volume element

[ProID fe_mat('p_solid','SI',1) Coordm In Stress Isop]

ProID	Property identification number.
Coordm	Identification number of the material coordinates system. **Warning** not implemented for all material formulations.
In	Integration rule selection (see integrules Gauss). 0 selects the legacy 3D mechanics element (of_mk_pre.c), -3 the default rule.
Stress	Location selection for stress output (NOT USED).
Isop	Integration scheme (will be used to select shear protection mechanisms).

The underlying physics for this subtype are selected through the material property. Examples are 3D mechanics with m_elastic, piezo electric volumes (see m_piezo), heat equation (p_heat).

2 : 2D volume element

[ProId fe_mat('p_solid','SI',2) Form N In]

ProID	Property identification number.
Type	Identifier obtained with fe_mat('p_solid,'SI',2).
Form	Formulation (0 plane strain, 1 plane stress, 2 axisymmetric), see details in m_elastic.
N	Fourier harmonic for axisymmetric elements that support it.
In	Integration rule selection (see integrules Gauss). 0 selects legacy 2D element, -3 the default rule.

The underlying physics for this subtype are selected through the material property. Examples are 2D mechanics with m_elastic.

3 : ND-1 coupling element

[ProId fe_mat('p_solid','SI',3) Integ Form Ndof1 ...]

ProID	Property identification number.
Type	Identifier obtained with fe_mat('p_solid','SI',3).
Integ	Integration rule selection (see integrules Gauss). 0 or -3 selects the default for the element.
Form	1 volume force, 2 volume force proportional to density, 3 pressure, 4: fluid/structure coupling, see fsc, 5 2D volume force, 6 2D pressure. 8 Wall impedance (acoustics), then uses the R parameter in fluid.

See also

Section 4.5.1, 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')
il=p_spring('dbval MatId Value')
il=p_spring('dbval -unit TM ProId name');
il=p_spring('dbval -punit 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 2 -unit TM 1e12') % Generate in TM, provide data in SI
il=p_spring(il,'dbval 2 -punit TM 1e9') % Generate in TM, provide data in TM
```

p_spring currently supports 2 subtypes

1 : standard

[ProID type k m c Eta S]

ProID	property identification number.
type	identifier obtained with fe_mat('p_spring','SI',1).
k	stiffness value.
m	mass value.
c	viscous damping value.
eta	loss factor.
S	Stress coefficient.

2 : bush

Note that type 2 is only functional with cbush elements.

[ProId Type k1:k6 c1:c6 Eta SA ST EA ET m v]

ProID	property identification number.
type	identifier obtained with fe_mat('p_spring','SI',2).
ki	stiffness for each direction.
ci	viscous damping for each direction.
SA	stress recovery coef for translations.
ST	stress recovery coef for rotations.
EA	strain recovery coef for translations.
ET	strain recovery coef for rotations.
m	mass.
v	volume.

See also

Section 4.5.1, section 7.4 , fe_mat, celas, cbush

p_super

Purpose

Element property function for superelements.

Syntax

```
il=p_super('default')
il=p_super('database MatId Value')
il=p_super('dbval MatId Value')
il=p_super('dbval -unit TM ProId name');
il=p_super('dbval -punit TM ProId name');
```

Description

If ProID is not given, fe_super will see if SE.Opt(3,:) is defined and use coefficients stored in this row instead. If this is still not given, all coefficients are set to 1. **Element property rows** (in a standard property declaration matrix il) for superelements take the forms described below with ProID the property identification number and coefficients allowing 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}.

Database,Dbval] ...

There is no database call for p_super entries.

1 : simple weighting coefficients

[ProId Type coef1 coef2 coef3 ...]

ProID	Property identification number.
Type	Identifier obtained with fe_mat('p_super','SI',1).
coef1	Multiplicative coefficient of the first matrix of the superelement (K{1}). Superelement matrices used for the assembly of the global model matrices will be {coef1*K{1}, coef2*K{2}, coef3*K{3}, ...}. Type of the matrices (stiffness, mass ...) is not changed. Note that you can define parameters for superelement using fe_case(model,'par'), see fe_case.

2 : matrix type redefinition and weighting coefficients

[ProId Type Form type1 coef1 type2 coef2 ...]

ProID	Property identification number.
Type	Identifier obtained with `fe_mat('p_super','SI',2)`.
type1	Type redefinition of the first matrix of the superelement (`K{1}`) according to SDT standard type (1 for stiffness, 2 for mass, 3 for viscous damping... see `fe_mknl MatType`).
coef1	Multiplicative coefficient of the first matrix of the superelement (`K{1}`). Superelement matrices used for the assembly of the global model matrices will be `{coef1*K{1}, coef2*K{2}, coef3*K{3}, ...}`. Type of the matrices (stiffness, mass ...) is changed according to type1, type2, ... Note that you can define parameters for superelement using `fe_case(model,'par')`, see `fe_case`.

See also

`fesuper`, section 6.3

p_piezo

Purpose

Property function for piezoelectric shells and utilities associated with piezoelectric models.

Syntax

```
mat= m_piezo('database name')
pl = m_piezo('dbval MatId  -elas 12 Name');
```

See section 6.1.5 for tutorial calls. Accepted commands are

ElectrodeMPC

`[model,InputDOF(end+1,1)]=p_piezo('ElectrodeMPC Name',model,'z==5e-5');` defines the isopotential constraint as a case entry `Name` associated with `FindNode` command `z==5e-5`. An illustration is given in section 6.1.5 .

Accepted command options are

- `-Ground` defines a fixed voltage constraint `FixDof,V=0 on` *Name*.
- `-Input"InName"` defines an enforced voltage `DofSet,`*InName* entry for voltage actuation.
- `MatId`*i* is used to define a resultant sensor to measure the charge associated with the electrode. Note that the electrode surface must not be inside the volume with `MatId`*i*. If that is the case, you must arbitrarily decompose your mesh in two parts with different `MatId`. You can also generate this sensor a posteriori using `ElectrodeSensQ`, which attempts to determine the `MatId`*i* based on the search of a piezoelectric material connected to the MPC.

ElectrodeSensQ

`model=p_piezo('ElectrodeSensQ 1682 Q-Base',model);` adds a charge sensor (`resultant`) called `Q-Base` on node `1682`. (See (1.4) for theory).

For **shells**, the node number is used to identify the `p_piezo` shell property and thus the associated elements. It is reminded that `p_piezo` entries must be duplicated when multiple patches are used. For **volumes**, the `p_piezo ElectrodeMPC` should be first defined, so that it can be used to obtain the electrode surface information.

Note that the command calls `fe_case('SensMatch')` so that changes done to material properties after this call will not be reflected in the observation matrix of this sensor.

To obtain sensor combinations (add charges of multiple sensors as done with specific wiring), specify a data structure with observation `.cta` at multiple `.DOF` as illustrated below.

For a voltage sensor, you can simply use a DOF sensor
`model=fe_case(model,'SensDof','V-Base',1682.21)`.

```
model=d_piezo('meshULBPlate cantilever');  % creates the model
% If you don't remember the electrode node numbers
p_piezo('ElectrodeDOF',model)
% Combined charge
r1=struct('cta',[1 1],'DOF',[1684;1685]+.21,'name','QS2+3');
model=p_piezo('ElectrodeSensQ',model,r1);
sens=fe_case(model,'sens');
% Combined voltage
r1=struct('cta',[1 1],'DOF',[1684;1685]+.21,'name','VS2+3');
model=fe_case(model,'SensDof',r1.name,r1);
sens=fe_case(model,'sens');sens.lab
```

ElectrodeDOF

`p_piezo('ElectrodeDof Bottom',model)` returns the DOF the bottom electrode. With no name for selection `p_piezo('ElectrodeDof',model)` the command returns the list of electrode DOFs based on MPC defined using the `ElectrodeMPC` command or `p_piezo` shell entries. Use `ElectrodeDof.*` to get all DOFs.

ElectrodeView ...

`p_piezo('electrodeview',cf)` outlines the electrodes in the model and prints a clear text summary of electrode information. To only get the summary, pass a model `model` rather than a pointer `cf` to a `feplot` figure.

`p_piezo('electrodeviewCharge',cf)` builds a `StressCut` selection allowing the visualization of charge density. You should be aware that only resultant charges at nodes are known. For proper visualization a transformation from charge resultant to charge density is performed, this is known to have problem in certain cases so you are welcome to report difficulties.

Electrode2Case

`Electrode2Case` uses electrode information defined in the obsolete `Electrode` stack entry to generate appropriate case entries : `V_In` for enforced voltage actuators, `V_Out` for voltage measurements, `Q_Out` for charge sensors.

ElectrodeInit

ElectrodeInit analyses the model to find electric master DOFs in piezo-electric shell properties or in MPC associated with volume models.

Tab

Tab commands are used to generate tabulated information about model contents. The calling format is p_piezo('TabDD',model). With no input argument, the current feplot figure is used. Currently generated tabs are

- TabDD constitutive laws
- TabPro material and element parameters shown as java tables.

View

p_piezo('ViewDD',model) displays information about piezoelectric constitutive laws in the current model.

p_piezo('ViewElec ...',model) is used to visualize the electrical field. An example is given in section 6.1.5 . Command options are DefLen*val* to specify the arrow length, EltSel*val* for the selection of elements to be viewed, Reset to force reinit of selection.

ViewStrain and ViewStress follow the same calling format.

Shell element properties

Piezo shell elements with electrodes are declared by a combination of a mechanical definition as a layered composite, see p_shell 2, and an electrode definition with element property rows of the form

```
[ProId Type MecaProId ElNodeId1 LayerId1 UNU1 ElNodeId2...]
```

- Type typically fe_mat('p_piezo','SI',1)
- MecaProId : ProId for mechanical properties of element p_shell 2 composite entry. The MatId*i* for piezo layers must be associated with piezo electric material properties.
- ElNodId1 : NodeId for electrode 1. This needs to be a node declared in the model but its position is not used since only the value of the electric potential (DOF 21) is used. You may use a node of the shell but this is not necessary.

- LayerId : layer number as declared in the composite entry.
- UNU1 : currently unused property (angle for polarization)

The constitutive law for a piezoelectric shell are detailed in section 6.1.5 . The following gives a sample declaration.

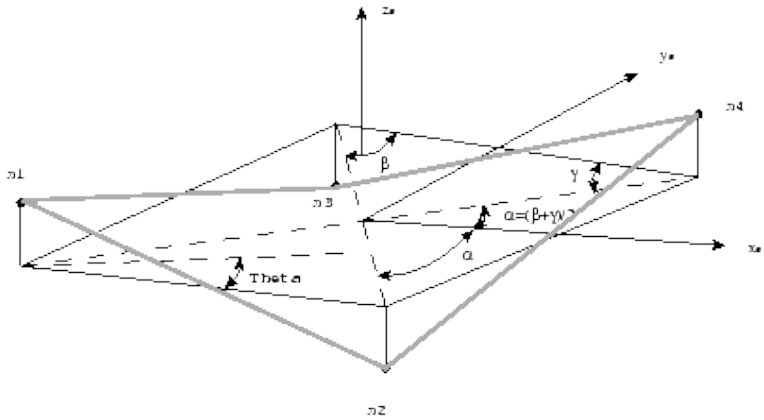
```
model=femesh('testquad4'); % Shell MatId 100 ProdId 110

% MatId 1 : steel, MatId 12 : PZT elastic prop
model.pl=m_elastic('dbval 1  Steel');
% Sample ULB piezo material, sdtweb m_piezo('sample_ULB')
model.pl=m_piezo(model.pl,'dbval 3 -elas 12 Sample_ULB');

% ProId 111 : 3 layer composite (mechanical properties)
model.il=p_shell(model.il,['dbval 111 laminate ' ...
    '3 1e-3 0 ' ...  % MatID 3 (PZT),   1 mm piezo, 0
    '1 2e-3 0  ' ... % MatID 1 (Steel), 2 mm
    '3 1e-3 0']);    % MatID 3 (PZT),   1 mm piezo, 0
% ProId 110 : 3 layer piezo shell with electrodes on nodes 1682 and 1683
model.il=p_piezo(model.il,'dbval 110 shell 111 1682 1 0 1683 3 0');

p_piezo('viewdd',model) % Details about the constitutive law
p_piezo('ElectrodeInfo',model) % Details about the layers
```

quad4, quadb, mitc4

Purpose

4 and 8 node quadrilateral plate/shell elements.

Description



In a model description matrix, **element property rows** for quad4, quadb and mitc4 elements follow the standard format

```
[n1 ... ni MatID ProID EltID Theta Zoff T1 ... Ti]
```

giving the node identification numbers ni (1 to 4 or 8), material MatID, property ProID. Other **optional** information is EltID the element identifier, Theta the angle between material x axis and element x axis, Zoff the off-set along the element z axis from the surface of the nodes to the reference plane (use feutil 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

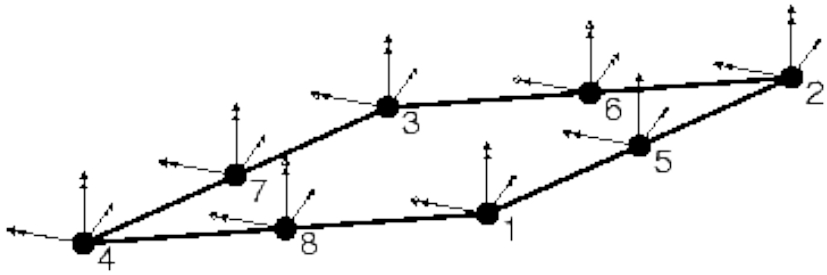
Supported formulations (f value stored in il(3) p_shell entries for isotropic materials and element default for composites) are

- 0 element/property dependent default. This is always used for composites (p_shell subtype 2).

- 5 Q4CS is a second implementation MITC4 elements that supports classical laminated plate theory (composites) as well as the definition of piezo-electric extension actuators. This is the default for SDT. Non flat shell geometries are supported with interpolation of normal fields.
- 1 4 tria3 thin plate elements with condensation of central node. Old and not very efficient formulation implemented in quad4.
- 2 Q4WT for membrane and Q4gamma for bending (implemented in quad4). This is only applicable if the four nodes are in a single plane. When not, formulation 1 is called.
- 4 MITC4 calls the MITC4 element below. This implementation has not been tested extensively, so that the element may not be used in all configurations. It uses 5 DOFs per node with the two rotations being around orthogonal in-plane directions. This is not consistent for mixed element types assembly. Non smooth surfaces are not handled properly because this is not implemented in the `feutil GetNormal` command which is called for each group of mitc4 elements.

The definition of local coordinate systems for composite fiber orientation still needs better documentation. Currently, q4cs the only element that supports composites, uses the local coordinate system resulting from the `BuildNDN 23` rule. A temporary solution for uniform orientation is provided with `model=feutilb('shellmap -orient dx dy dz',model)`.

quadb



Supported formulations (p_shellil(3) for isotropic materials and element default for composites) are

- 1 8 tria3 thin plate elements with condensation of central node.
- 2 isoparametric thick plate with reduced integration. For non-flat elements, formulation 1 is used.

See also

m_elastic, p_shell, fe_mk, feplot

q4p, q8p, t3p, t6p and other 2D volumes

Purpose

2-D volume elements.

Description

The q4p q5p, q8p, q9a, t3p, t6p elements are topology references for 2D volumes and 3D surfaces.

In a model description matrix, **element property rows** for this elements follow the standard format

```
[n1 ... ni MatID ProID EltID Theta]
```

giving the node identification numbers n1,...ni, material MatID, property ProID. Other **optional** information is EltID the element identifier, Theta the angle between material x axis and element x axis (material orientation maps are generally preferable).

These elements only define topologies, the nature of the problem to be solved should be specified using a property entry, see section 6.1 for supported problems and p_solid, p_heat, ... for formats.

Integration rules for various topologies are described under integrules. Vertex coordinates of the reference element can be found using an integrules command containing the name of the element such as r1=integrules('q4p');r1.xi.

Backward compatibility note : if no element property entry is defined, or with a p_solid entry with the integration rule set to zero, the element defaults to the historical 3D mechanic elements described in section 7.19.2 .

These volume elements are used for various problem families.

See also

fe_mat, fe_mk, feplot

rigid

Purpose

Linearized rigid link constraints.

Description

Rigid links are often used to model stiff connections in finite element models. One generates a set of linear constraints that relate the 6 DOFs of master M and slave S nodes by

$$\begin{Bmatrix} u \\ v \\ w \\ r_x \\ r_y \\ r_z \end{Bmatrix}_S = \begin{bmatrix} 1 & 0 & 0 & 0 & z_{MS} & -y_{MS} \\ 0 & 1 & 0 & -z_{MS} & 0 & x_{MS} \\ 0 & 0 & 1 & y_{MS} & -x_{MS} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \\ r_x \\ r_y \\ r_z \end{Bmatrix}_M \tag{9.25}$$

Resolution of linear constraints is performed using `fe_case` or model assembly (see section 4.8.8) calls. The theory is discussed in section 7.14 . Note that the master node of a rigid link has 6 DOF, even if the model may only need less (3 DOF for volumes).

If coordinate systems are defined in field `model.bas` (see `basis`), `PID` (position coordinate system) and `DID` (displacement coordinate system) declarations in columns 2 and 3 of `model.Node` are properly handled.

Although `rigid` are linear constraints rather than true elements, such connections can be declared using an element group of rigid connection with a header row of the form `[Inf abs('rigid')]` followed by as many element rows as connections of the form

```
[ n1 n2 DofSel MatId ProId EltId]
```

where node `n2` will be rigidly connected to node `n1` which will remain free. `DofSel` lets you specify which of the 3 translations and 3 rotations are connected (thus `123` connects only translations while `123456` connects both translations and rotations). The rigid elements thus defined can then be handled as standard elements.

With this strategy 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 and set a stiffness constant `Kv`, one can do

```
model=feutil('SetGroup rigid name celas',model);
model.Elt(feutil('findelt group i',model),7) = Kv; % celas in group i
```

The other rigid definition strategy is to store them as a case entry. rigid entries are rows of the Case.Stack cell array giving {'rigid', Name, Elt}.

The syntax is

```
model=fe_case(model,'rigid',Name,Elt);
```

where Name is a string identifying the entry. Elt is a model description matrix containing rigid elements. Command option Append allows concatenating a new list of rigid constraints to a preexisting list in Case.Stack.

The call model=fe_case(model,'rigidAppend','Name',Elt1); would thus concatenate the previously defined list Name with the new rigid element matrix Elt1.

Using the fe_case call to implement rigid allows an alternative rigid constraint input that can be more comprehensive in some applications. You may use a list of the form [MasterNode slaveDOF slaveNode_1 slaveNode_2 ... slaveNode_i] instead of the element matrix. Command option Append is also valid.

The following sample calls are thus equivalent, and consists in implementing a rigid link between nodes 1 and 2, and 1 and 3 (with 1 as master) for all six DOF in a sample model:

```
model=fe_case(model,'rigid','Rigid edge',...
[Inf abs('rigid');
1 2 123456 0 0 0;
1 3 123456 0 0 0]);
% or
model=fe_case(model,'rigid','Rigid edge',[1 123456 2 3]);
```

In some cases, interactions with feplot visualization may transform the Elt matrix into a structure with fields Elt that contains the original data, and Sel that is internally used by feplot to display the rigid constraint on the mesh.

The following example generates the mesh of a square plate with a rigid edge, the rigid constraint is here declared as rigid elements

```
% generate a sample plate model
model=femesh('testquad4 divide 10 10');

% generate beam1 elements based on the edge
% of the underlying 2D model at x=0
elt=feutil('selelt seledge & innode{x==0}',model);
% remove element header from selection,
```

```
% we only use the node connectivity
elt=elt(2:end,:);
% assign the rigid element property
elt(2:end,3)=123456; % all 6 DOF are slave
% remove old data from the previous element selection
elt(2:end,4:end)=0;

% add rigid elements to the model
model=feutil('addelt',model,'rigid',elt);
% % alternative possible: define as a case entry
% model=fe_case(model,'rigid','Rigid edge',[Inf abs('rigid'); elt]);

% Compute and display modes
def=fe_eig(model,[6 20 1e3]);
feplot(model,def);fecom(';view3;ch8;scd.1');
```

The `rigid` function itself is only used for low level access by generating the subspace `T` that verifies rigid constraints

```
[T,cdof] = rigid(node,elt,mdof)
[T,cdof] = rigid(Up)
```

See also

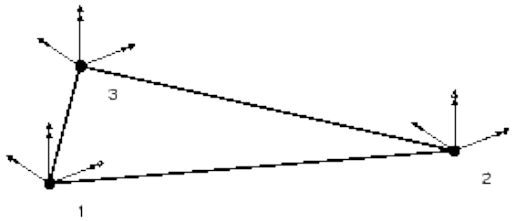
Section 7.14, `celas`

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 : p_shellformulation is not used, the single implemented formulation uses a T3 triangle for membrane properties and a DKT for bending (see [46] for example).

tria6 :p_shellformulation is not used since the currently the only implementation is a call to q4cs (formulation 5).

See also

quad4, quadb, fe_mat, p_shell, m_elastic, fe_mk, feplot

Function reference

abaqus 460
ans2sdt 471
basis 475
comgui,cingui 479
commode 489
comstr 491
curvemodel 495
db, phaseb 497
ex2sdt 498
fe2ss 501
fecom 505
femesh 524
feutil 538
feutila 570
feutilb 571
feplot 589
fesuper 594
fjlock 603
fe_c 609
fe_case 612
fe_caseg 627
fe_ceig 631
fe_coor 633
fe_curve 635
fe_cyclic 643
fe_def 647
fe_eig 650

fe_exp 654
fe_gmsh 658
fe_load 663
fe_mat 668
fe_mknl, fe_mk 671
fe_norm 677
fe_quality 679
fe_range 685
fe_reduc 698
fe_sens 703
fe_simul 709
fe_stress 711
fe_time 716
of_time 728
idcom 732
idopt 737
id_dspi 740
id_nor 742
id_poly 746
id_rc, id_rcopt 747
id_rm 751
iicom 754
iimouse 766
iiplot 770
ii_cost 774
ii_mac 775
ii_mmif 787
ii_plp 794
ii_poest 799
ii_pof 801
lsutil 803
nasread 805
naswrite 810
nor2res, nor2ss, nor2xf 816
of2vtk 824
ofact 826

perm2sdt **831**
polytec **833**
psi2nor **836**
qbode **838**
res2nor **840**
res2ss, ss2res **841**
res2tf, res2xf **844**
rms **845**
samcef **846**
sd_pref **848**
setlines **849**
sdtcheck **850**
sdtdef **852**
sdth **855**
sdthdf **856**
sdtroot, sdt_locale **861**
sdtweb **864**
sp_util **866**
stack_get,stack_set,stack_rm **868**
ufread **870**
ufwrite **876**
upcom **878**
up_freq, up_ifreq **887**
up_ixf **888**
v_handle **889**
xfopt **890**

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 9.

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

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_stress	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
sdtcheck	installation handling and troubleshooting utilities

abaqus

Purpose Interface between ABAQUS and SDT (part of FEMLink) **Warning this function requires MATLAB 7.1 or later.**

Syntax

```
abaqus('read FileName');
abaqus('job');
```

`read[*.fil, *.inp, *.mtx, *.dat]`

By itself the `read` command imports the model from a `.inp` ASCII input or `.fil` binary output file. Support for `.dat` read is very partial, but provides a framework for users to parse desired tokens.

Models created by an `*Assembly` command using several instances and/or additional nodes or elements are treated with superelements. Each part instance (called by `*Instance...*end instance`) becomes then a specific superelement in the SDT model. A packaged call allows to get a full model back

```
model=abaqus('read Job-1.inp');
model=abaqus('ResolveModel',model);
% both calls at once:
model=abaqus('read-resolve Job-1.inp');
```

The `ResolveModel` command has a limited robustness in the general case due to the difficulty to handle heterogeneous Stack data while renumbering parts of a model. Most cases should be properly handled. One can use command `read-resolve` to perform both operations at once.

When reading deformations, `sdtdef('OutOfCoreBufferSize')` is used to determine whether the vectors are left in the file or not. When left, `def.def` is a `v_handle` object that lets you access deformations with standard indexing commands. Use `def.def=def.def(:,:)` to load all. If a modal basis is read, it is stored in the model stack, as `curve,Mode`. If static steps are present all associated deformation are concatenated in order of occurrence in the model stack as `curve,step(1)`.

Command option `-wd` allows to save the model generated in a directory different from the one in which the abaqus files are saved.

You can request the output of element matrices which will then be read into an `upcom` model. To do so, you need to define an element set. To read matrices, you have to provide some information before running the job in order to select which matrices you want to write and read. In the `.inp` input file you may enter the following line

```
*ELSET, ELSET=ALL ELT FOR SDT
THIN SHELL1 , THIN SHELL1_1
```

(second line contains all the ABAQUS defined sets) just before the *STEP line and

```
*ELEMENT MATRIX OUTPUT, ELSET=ALL ELT FOR SDT, STIFFNESS=YES
*ELEMENT MATRIX OUTPUT, ELSET=ALL ELT FOR SDT, MASS=YES
```

just after the *STEP line.

Note that this information are automatically generated using the following command `abaqus('elementmatrices model.inp');` .

Running the Abaqus job generates outputs specified by the user, with *OUTPUT commands in the Abaqus job input file. Current default use generates an odb file, using commands of the type *NODE OUTPUT. The odb format however requires the use of Abaqus libraries to be read.

Imports are thus handled in SDT using the .fil output binary file. This file is readable without Abaqus, and its reading has been optimized in FEMLink. This type of output is generated using commands of the type *NODE FILE. A sample command to obtain nodal deformation a the end of a step is then

```
** general command to .fil and ask for nodal deformation field
*OUTPUT, FIELD
*NODE FILE
 U
```

All nodal variable keywords should be expressed on separated lines. This must be repeated in all steps of interest in an ABAQUS computation file input .inp.

Most common and general nodal variables keywords of interest are the following (this is not applicable to all ABAQUS procedures)

- U, V, A respectively for nodal displacement, velocity and acceleration output
- RF,CF, VF, TF respectively for nodal reaction forces, constrained forces, viscous forces, and total forces output
- GU, GV, GA respectively for generalized displacement, velocity ad acceleration (when reduction is involved)

Since not all information (materials, set names, ...) can be found in the .fil, you may want to combine two reads into an upcom model

`abaqus('read file.inp', 'buildup file.fil');`.

Abaqus features a matrix sparse output starting from version 6.7-1. Their generation is performed in a dedicated step as follows

```
*STEP
*MATRIX GENERATE, STIFFNESS, MASS
*END STEP
```

The output is one ASCII file `.mtx` by matrix requested, which can be read by `abaqus`.

Reading a `.dat` file should be avoided in general as the ASCII storage format and variation between ABAQUS versions makes it unpractical. There are however cases where such reading is the easiest way; A framework adapted to such parsing is provided with support to read complex mode shapes (that cannot be stoed in the `.fil` file).

One can call `data=abaqus('Read',fdat,li)`; with `fdat` a `.dat` file and `li` an optionnal Nx2 cell array providing a list of tokens to detect and and associated callback. The supported tokens are used if `li` is omitted, it is separately accessible with `li=abaqus('DatList');` if users wish to combine supported features with customized ones.

If a token is detected in the file, a callback will be fired as `out1=feval(cbk1,fid,evt,cbk2:end)`; with `cbk` the callback cell array provided in the second column of `li`, `fid` the valid opened file object set a the starting position of the currently detected token, `evt` a structure with fields `.p0` the starting position of the scanned text buffer (not the current position to be recovered by `pcur=ftell(fid);`, `.p1` the file length, `.bufs` a buffer size to be exploited. The callback command must rethrow a structure whose field will be incremently added to the global output structure.

Build[model,case,contact]

Thise set of high level commands aims at transforming a raw imported model into a functional model in SDT. It exploits in particular the lower level `abaqus Resolve` commands.

- `BuildCase step` *`istep`*
  `model=abaqus('BuildCase step1',model);` This command prepares the model case loading corresponding to a given step index *`istep`*. Raw model reading imports indiferently all boundary conditions and loading into the Case Stack. The loading sequence is stored in the stack entry `info,BSHist` and is exploited by `BuildCase` to generate the loading relative to a given step. One can ask for the last step by using token *`steplast`* instead of `step` *`istep`*. Command options
  - `all` restores all case entries in the Case Stack.

  - -noResolve asks not to perform the abaqus Resolve call if this was previously performed.

- BuildModel step *istep*
  model=abaqus('BuildModel step1',model); This command generates the model global state at a given step specified by step index *istep*. In addition to the BuildCase functionalities, this function looks for a static response result corresponding to the given set to define a curve,q0 entry, thus declaring a static state in the SDT model. One can ask for the last step by using token *steplast* instead of step *istep*. Command options

  - -noResolve asks not to perform the abaqus Resolve call if this was previously performed.
  - -contact*CAM* link to a call to the BuildContact command with forwarding of additionnal command options given in *CAM*. **This feature is only accessible with a valid SDT-Contact module license**.
  - -getStatic to only resolve the static state. The output is then the static state. It is possible to specifiy in intial set of static deformations in an additional argument. q0=abaqus('buildModel-steplast-getStatic',model,... stack_get(mo1,'curve','step(1)','get')).

- BuildContact step *istep*
  model=abaqus('BuildContact step1',model); This command packages the generation of SDT contact elements and laws based on the ABAQUS definition. **This feature is only accessible with a valid SDT-Contact module license**. The import generates contact elements based on master surfaces with penalized contact laws. *Hard contact* laws are thus automatically penalized with a calibrated stiffness density. Support for the *CONTACT PAIR, *MOTION, *CLEARANCE, *CHANGEFRICTION commands is provided and integrates *SURFACE BEHAVIOR and *FRICTION law inputs.

  The step definition is mostly usefull for *MOTION and *CHANGE FRICTION commands.One can ask for the last step by using token *steplast* instead of step *istep*. Command options

  - -module has to be used for users with no access to the SDT-NL tools outside SDT-Contact.
  - Command option -useRes asks to initialize contact states based on static force resultants on surface rather than by observing gaps on the static deformation field. This can be usefull to alleviate contact state import discrepancies due to different contact implementations between ABAQUS and SDT.
  - -optim is used to remove curves from the model that are not usefull for further anlysis after the BuildContact step is performed.
  - -tgStickNoMotion can be used to define tangential sticking property for contact with friction and no motion.

- `-moRot` is used to specify the local definition of contacts : tangent and normal directions
  - `-moRot''cyl''` defines a cylindrical contact (for example the sliding contact of a drum brake)

`Resolve`

This set of commands transforms a raw model import by `abaqus read` into en exploitable SDT model. This is for example useful when the ABAQUS model has been generated with `*PART` and `*INSTANCE`. In such case, the representation of an ABAQUS model becomes very far from an SDT model. The raw reading obtained by `read` will thus interpret parts as superelements, and leave the instance data, and some internal information not translated. Some other advanced definitions need special care and are thus handled in this section.

Some adaptations, performed by `ResolveModel` are thus needed. In particular, renumbering can occur, however all sets definitions are maintained.

- `ResolveModel`
  This command will create the elements conforming to the instance information. Commands `ResolveSet`, `ResolveMass`, `AssembleUserElements`, `ResolveCase` and `ResolveShellC` will also be called, to generate a fully exploitable SDT model.
- `ResolveSet`
  This command transforms each ABAQUS implicitly defined sets into explicit SDT sets. This is very useful if some sets have been defined in ABAQUS using internal part numerotations. This command is also usefull to distinguich sets of different types but with initially the same name. This behavior is not available in SDT and special care is taken not to mixup set names and types. Called by `ResolveModel`.
- `ResolveCase`
  This command aims at resolving all implicitly defined case entries in the model, including `*MODEL CHANGE`, and some connector calls. This also handles the multiple slave resolution in the manner of ABAQUS, and should thus be performed before assembling models if multiple slave error occur.
- `ResolveMass`
  This command handles the model stack entry `info,UnResolvedMasses` that may have been created during the `read` call, and assigns mass values missing in mass elements. This is necessary when masses have been defined in an ABAQUS part, such that the attribution of the mass amplitude by `*MASS` is not directly retrievable. Called by `ResolveModel`.

- ResolveShellC
  Continuum shell elements (SC8R and SC6R) have no direct counterparts in SDT. A base resolution just ignores the shell declaration and declare these elements as solids with reduced integration (this may not work for stacked layers of continuum shells). The following command options are available
  - -shellSE will generate a superelement embedding shells in SDT format from the neutral fiber of the continuum shells, within the 3D topology. In that way a behavior equivalent to ABAQUS is expected.
  - -order2 in combination with -shellSE uses second ordr shells instead of first order ones.

write

abaqus('write Name.inp',model); writes and ABAQUS input file.

abaqus('BwMTX',model); writes all matrices stored in model.K in the abaqus sparse output format. Each matrix file is named after the model.file entry and model.Klab. For a model stored in model.mat containing a matrix 'k', the file output will be named model_k.mat.

BwMat ; BwMp ; BwSet ; Bwbas ; BwStepEig are implemented.

JobOpt

JobOpt = abaqus('JobOpt',Opt); This command returns a filled JobOpt structure to be run by sdtjob. Opt is a structure containing at least the field Job as the job name or file. InList and OutList must be filled. Further options concern the fields Input when the input file is different from the job name, RunOptions to append the usual option to the Abaqus command, RemoveFile to remove files from the remote directory when needed.

conv

This command lists conversion tables for elements, topologies, face topologies. You can redefine (enhance) these tables by setting preferences of the form
sd_pref('set','FEMLink','abaqus.list',value), but please also request enhancements so that the quality of our translators is improved.

splitcelas

`model=abaqus('SplitCelas',model)` splits all `SDT celas` elements to one dimension `celas` elements that can be handled by Abaqus. This command can change the `EltId` so it must be used when meshing the model.

uniquematpro

Merges duplicated `pl`/`il` instances.

AssembleUserElements

Returns a matrix and its corresponding DOF, from the assembly of all USER ELEMENT instances in an ABAQUS model. This command is exploited in `abaqus Resolve` calls.

```
[K,dof] = abaqus('AssembleUserElements',model);
```

Command option `-inModel` directly sets a SDT functional superelement named `usere` in the model. In this case, element matrices are removed from the stack. They can be kept with command option `-keep`.

Command option `-disjsplit` splits the assembled SE into disjoint SE regarding DOF connectivit, resulting SE are named `ue`i with i a 6 digit fixed index.

```
model=abaqus('AssembleUserElements-inModel',model);
```

odb2sdt

Utility functions to transfer Abaqus `.odb` file data into a format similar to MATLAB 6 binary `.mat` file and readable by `sdthdf`. The changes in the format are introduced to support datasets larger than 2GB.

Abaqus outputs are commonly written in `.odb` files, using a non documented format. The only way to access its data is to use Abaqus CAE or Abaqus Python. These utility functions are to be used with Abaqus Python to extract data from the output database for further use outside Abaqus. The modules used are

- odbAccess. Abaqus access libraries.
- abaqusConstants. Common output values dictionary, such as `'U'`, `'UR'`
- Numeric. Module for array handling utilities.
- struct. Module to pack data into binary strings.

For the moment, only nodal data transfer is completely implemented. More information can be found on Python at http://www.python.org. Note that def is a reserved word in Python for the function definition command; remember not to use it in another way!

The following script is a quick example of what can be done with these functions. It can be launched directly if written in a .py file readODB.py for example, by abaqus python readODB.py

```
from odb2sdt import *   # import read functions

jobName='my_abaqus_job'
odb=openOdb(jobName + '.odb')
allNodal2mat(odb)
```

This second script will only write the DOF set in a .mat binary file

```
from odb2sdt import *   # import read functions

jobName='my_abaqus_job'

odb=openOdb(jobName + '.odb')       #open the database
stepName=odb.steps.keys()[0]        #get the name of the first step
fieldItem=['U']                     #I want the 'U' displacement field

# get the fieldOutputs instances list from the first frame:
fieldOutputs=odb.steps.__getitem__(stepName).getFrame(0).fieldOuputs

f=matFile(jobName + '_dof.mat')              # Initialize the file
dof2mat(f,fieldOutputs,fieldItem,stepName)   # write the DOF array to it
f.close()
```

Once a file_allNodal.mat file has been generated, it is possible to load the deformation structure fields using

```
def=abaqus('read file_allNodal.mat')
```

def output is here a cell array containing all def structures found in the allNodal.mat file. Only simple cases of .odb outputs are supported. The rest of the data is not automatically read, it can nevertheless be attained using

```
r1=sdthdf('open',file_allNodal.mat);
```

where r1 is a cell array containing all the fields contained in the allNodal.mat file.

odb2sdt.py reference

The following lists the main subfunctions in odb2sdt.py

matFile(fname)	Creation of a the file fname, with the standard .mat header.f=matFile(fname)
dof2mat(f, fields ,fieldItems, stepName)	Writes the DOF array in SDT format to file f. fields is the list of fieldOutput instances from the step named stepName. fieldItems is the sorted list containing the displacement fieldOutputs present in the fieldOuputs list. It must contain in that order, and at least one entry of the list ['U' , 'UR' , 'UT']. It is a direct call with no output.
defSet2mat(f, step, fieldList)	Writes a fieldOutput set for all frames of a step, contiguously into file f. step is a step instance, fieldList is the list of fieldOutputs to be output from the frame object. All kind of nodal vector output can be treated although this was designed to treat displacement fields linked to the dof2mat function. It is a direct call with no output. In case of a modal deformation set, the EIGIMAG, EIGFREQ, EIGREAL and DAMPRATIO historyOutput data are also output.
nodalScalarValues2mat (f, field, stepName, frameName)	Outputs an array of scalar nodal values to file f, for a particular fieldOutput instance field. stepName is the name of the step considered, frameName the name of the frame. However, since the fieldOutput is given the last two arguments are strings only needed to compose the array name in f.It is a direct call with no output.
allNodal2mat(odb)	This function combines the lower level nodal output function to create and fill directly a .mat file containing DOFs, deformations sets, and nodal scalar values form an odb instance, created with openOdb. It is a direct call with no output.

The following are lower level calls, and alternative calls, with output in the workspace.

`sortFieldList( fieldList)`	Returns a field keys list in which the existing displacement field keys have been sorted at the list beginning, in the order `'U'`, `'UR' , 'UT'`. `fieldList=sortFieldList(fieldList)`.
`rmFromList(list1, list2)`	Returns `list1` in which the items in `list2` have been removed.
`arrayHead2mat(f, nValSize, isCpx, dim1,dim2, arrayName)`	Low level command. Initialization of an array entry into the file `f`. The corresponding header is written such that the array values can be written right after. `nValSize` is the space needed to store the values form the array in Bytes. `isCpx` takes the value `0` if the data to store are real, or `16` if the values to store are complex. `dim1` and `dim2` are the dimensions of the array in direction 1 and 2. `arrayName` is the name given to the array. It is a direct call with no output.
`getNodes(frame)`	Returns a nodeId array in the workspace, taken in a frame instance.`nodeId=getNodes(frame)`
`getLabels(frame, fieldKeys)`	Returns the list of componentLabels contained in all the `fieldKeys` list, in a frame instance. It also generates a list in which the field keys are repeated to match the componentLabels list. `labels,labelField=getLabels(frame,fieldKeys)`
`setDOF(nodeId, field, fieldKeys)`	Returns a DOF array interpreted from a `fieldOuputs` list, a `nodeId` array and `fieldKeys` giving the fieldOutput displacement keys relevant in `field`. `DOF=setDOF(nodeId,fieldOutputs,['U'])`
`readData(value)`	A way to output a data member of a value instance regardless of the precision used during the computation. `data=readData(value)`
`readNodalValues(field, outList)`	Returns optionally the nodeId array, the corresponding data array and the componentLabels lists found, from a fieldOutput instance. OutList is a list of length 3 being [1,1,1] for a complete output, [0,1,0] to output only the data array, and [1,1,0] to output the combo nodeId array and data array. `nodeId,data=readNodalValues(fieldOutput, [1,1,0])`

Examples

See also FEMLink

ans2sdt

Purpose

Interface between ANSYS and SDT (part of FEMLink)

Syntax

```
ans2sdt('read FileName')          % .rst, .cdb, .matrix, .mode  files
ans2sdt('write FileName')         % .cdb file
ans2sdt('BuildUp FileName')       % .rst and .emat files
... = ans2sdt('def FileName.rst')% .rst or .mode files
```

Description

Build[Up,ContactMPC]

- Command BuildUp reads the binary files FileName.rst for model definition and FileName.emat for element matrices. The result is stored in Up (a type 3 superelement handled by upcom). FileName.mat is used to store the superelement.

  General syntax is ans2sdt('BuildUp FileName'); valid calls are

  ```
  Up=ans2sdt('buildup file');
  [m,k]=upcom(Up,'assemble not');
  ```

  For recent versions of ANSYS, you will 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.

  There is a partial attempt to fill in element properties in Up.il. You can also use data=stack_get(model,'info','RealConstants','getdata') to obtain the cell array containing the ANSYS real constants for various elements. The index in this cell array corresponds to element ProId values.

- Command BuildContactMPC interprets ANSYS contact elements (CONTA171-175), and slave elements TARGE170 to generate MPCs in the form of fe_caseg ConnectionSurface. This is thus close to bonded contact formulations.

  ```
  model=ans2sdt('Read file.cdb'); % read base file
  % transform contact info into bonded coupling
  model=ans2sdt('BuildContactMPC',model);
  ```

def

`def=ans2sdt('read','file.mode')` reads deformations in `.mode` files.

To read responses `.rst` files you should use

```
model=ans2sdt('readdef','test.rst'); % read all data
def=stack_get(model,'curve','NSL');
% Partial read of only specific entries
model=ans2sdt('rstdef','sdtforced.rst', ...
  struct('DefUse',{{'NSL'}})); % give the block names to be read
```

Since multiple blocks can be read, the results is saved in the model stack and can be retrieved by name using `stack_get(model,'curve','NSL');` or similar calls. The standard names used by ANSYS are `NSL` (displacement), `VSL` (velocity response), `RF` (reaction forces), `ESL` (element solution, see `ans2sdt ESLread`). If you are interested in reading other results, please send a test case.

conv

This command lists conversion tables for elements, topologies, face topologies. You can redefine (enhance) these tables by setting preferences of the form
`sd_pref('set','FEMLink', 'ansys.elist',value)`, but please also request enhancements so that the quality of our translators is improved.

read

This command reads files based on their standard ANSYS extension.

- `.matrix` files are read assuming ASCII Harwell Boeing format obtained with `HBMAT, Fname,Ext,--,ASCII,STIFF`. RHS vectors or binary matrices are not read yet. You can read the mapping file at the same time using `ans2sdt('matrix','k.txt','k.mapping');` or `DOF=ans2sdt('mapping','k.mapping')`.
- `.mode` files contain deformations which are read into the usual SDT format.
- `.rst` files contains model information topology, some material/element properties and boundary conditions (but these are more consistently read in the `.cdb`), ...
  - When an `.emat` file is present, the read call attempts to run the `BuildUp` command.
  - Responses are read using a call of the form `ans2sdt('readdef','file.rst')`, see `ans2sdt def`

- .cdb input files also written by ANSYS using the CDWRITE ALL,FileName,cdb command. Please also request enhancements on the support of this format so that the quality of our translators is improved.

ANSYS does not store boundary conditions in the .rst files so that these can only be imported from .cdb file. If you only have fixed boundary conditions, you can easily generate those with

```
model=ans2sdt('buildup test');          % read model
def=ans2sdt('def test.rst');            % read deformations
model = fe_case(model,'fixdof','Fixed_Dofs', ...
  fe_c(model.DOF,def.DOF,'dof',2));
cf=feplot; cf.model=model; cf.def=def; % display
```

Def

def=ans2sdt('def FileName.rst') or def=ans2sdt('def FileName.mode') reads deformations in .rst or .mode files

ESLread

To read element output data if any, that were detected during the reading of an output file (.rst).

model=ans2sdt('ESLread'',model); will generate a stack entry named ESL:*token* in the model that will contain the element data.

token is an element output data identifier as documented by ANSYS, and mentioned in the model stack entry info,ptrESL.

Command option group*i* allows generating the output for a given group number *i*

JobOpt

JobOpt = ans2sdt('JobOpt',Opt); This command returns a filled JobOpt structure to be run by sdtjob. Opt is a structure containing at least the field Job as the job name or file. InList and OutList must be filled. Further options concern the fields Input when the input file is different from the job name, RunOptions to append the usual option to the Ansys command, RemoveFile to remove files from the remote directory when needed.

Write

`ans2sdt('write FileName.cdb',model)` is the current prototype for the ANSYS writing capability. In ANSYS `.cdb` files are written with the `CDWRITE ALL, FileName, cdb` command. This does not currently write a complete `.CDB` file so that some manual editing is needed for an ANSYS run after the write.

See also

`FEMLink`

basis

Purpose

Coordinate system handling utilities

Syntax

```
p          = basis(x,y)
[bas,x]    = basis(node)
[ ... ]    = basis('Command', ... )
```

Description

`nodebas [nodeGlob,bas]=basis('nodebas',model)`

`NodeBas` performs a *local to global node transformation* with recursive transformation of coordinate system definitions stored in `bas`. Column 2 in `nodeLocal` is assumed give displacement coordinate system identifiers `PID` matching those in the first column of `bas`. `[nodeGlobal,bas]=basis(nodeLocal,bas)` is an older acceptable format. `-force` is a command option used to resolve all dependencies in `bas` even when no local coordinates are used in `node`.

Coordinate systems are stored in a matrix where each row represents a coordinate system using any of the three formats

```
% different type of coordinate defintition
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.

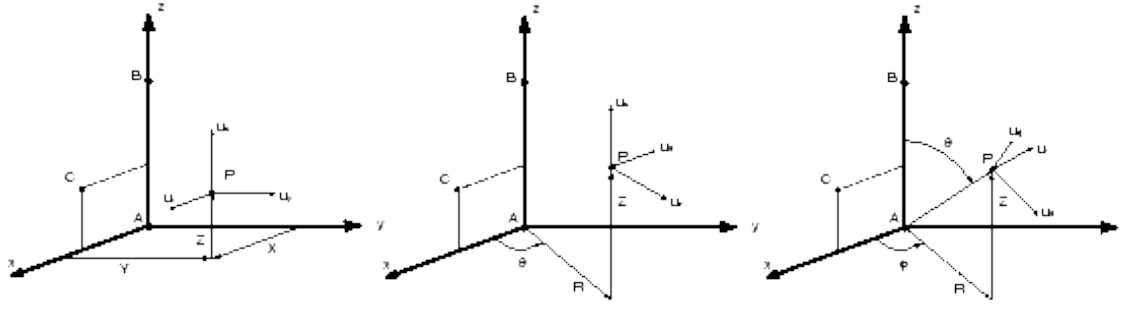


Figure 10.1: Coordinates convention.

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 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. When storing these vectors as columns one thus builds the $x_G = [c_G L]\, x_L$ transform.

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.

```
trans[ ,t][ ,l][,e] cGL= basis('trans [ ,t][ ,l][,e]',bas,node,DOF)
```

The *transformation basis for displacement coordinate systems* is returned with this call. Column 3 in `node` is assumed give displacement coordinate system identifiers `DID` matching those in the first column of `bas`.

By default, `node` is assumed to be given in global coordinates. The `l` command option 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 translations and 3 rotations at each node. The `t` command option restricts the result to translations. With the `DOF` argument, the output is defined at DOFs in `DOF`.

The `e` command option (for *elimination*) returns a square transformation matrix. **Warning:** use of the `transE` command and the resulting transformation matrix can only be orthogonal for translation DOF if all three translation DOF are present.

```
gnode: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)`.

`bas` can be specified as a string compatible with a `basis('rotate'` call. In such case, the actual basis is generated on the fly by `basis('rotate')` before applying the node transformation.

`[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.

`rotate`

`bas=basis('rotate',bas,'command',basId);` is used to perform rotations on coordinate systems of `bas` given by their `basId`. `command` is a string to be executed defining rotation in degrees (`rx=45;` defines a 45 degrees rotation along x axis). One can define more generally rotation in relation to another axis defining angle `r=angle` and axis `n=[nx,ny,nz]`. It is possible to define translations (an origin displacement) by specifying in `command` translation values under names `tx`, `ty` and `tz`, using the same formalism than for rotations.

For example, one can define a basis using

```
% Sample basis defintion commands
bas=basis('rotate',[],'rz=30;',1); % 30 degrees / z axis
bas=basis('rotate',[],'r=30;n=[0 1 1]',1); % 30 degrees / [0 1 1] axis
bas=basis('rotate',[],'tx=12;',1); % translation of 12 along x
bas=basis('rotate',[],'ty=24;r=15;n=[1 1 1];',1); % trans. of 24 along y and rot.
```

`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

$$p = \left[\frac{\vec{x}}{\|\vec{x}\|}, \frac{\vec{y}_1}{\|\vec{y}_1\|}, \frac{\vec{x} \times \vec{y}_1}{\|\vec{x}\|\|\vec{y}_1\|} \right] \tag{10.1}$$

where $\vec{y}_1$ is the component of $\vec{y}$ that is orthogonal to $\vec{x}$

$$\vec{y}_1 = \vec{y} - \vec{x} \frac{\vec{x}^T \vec{y}}{\|\vec{x}\|^2} \tag{10.2}$$

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`

```
% Generation of an orthonormal matrix
p =  basis([2 0 0],[1 1 1])

p =
     1.0000         0         0
          0    0.7071   -0.7071
          0    0.7071    0.7071
```

See also

`beam1`, section 7.1 ,section 7.2
Note : the name of this function is in conflict with `basis` of the *Financial Toolbox*.

comgui,cingui

Purpose

General utilities for graphical user interfaces and figure formatting. Figure formatting documentation can be found in section 8.1.

Syntax

```
comgui('Command', ...)
cingui('Command', ...)
```

comgui is an open source function that the user is expected to call directly while cingui is closed source and called internally by SDT.

ImCrop

Image cropping utilities. This function allows cropping uniform borders and uniform rows or columns in an image.

Syntax is a=comgui('ImCrop',a)'

Image a can be either

- an image defined by an m-by-n-by 3 matrix, or a line cell array of such images
- a structure from getFrame with fields cdata containing m-by-n-by 3 matrices
- a file name or a line cell array of file names. By default if a file name is given the file is replaced by saving the cropped image.
- a composite cell array line with file names and images. By default if a file name is given the file is replaced by saving the cropped image.

The following command options are available

- Borders To only crop image from the first border.
- AllBorders To only crop image from all borders.
- BorderNum To only crop image from the first N borders, given as parameter.
- UpToBorder To crop until a border is found in the limit of 20 pix from the edges of the original image. (Useful for java capture of figures)

- `All` To remove all rows/columns with equal colors throughout the image.
- `Equal` To apply the same cropping to all images in the cell array input, by intersecting cropping rows and columns.
- `-noSave` Not to erase images provided in file names.
- `Rot90` can be used to rotate the image by ± 90 degrees before cropping

You can include cropping options within an `ImWrite` call by defining a `.CropOpt` field in the option structure.

`ImWrite, ...`

`ImwriteFileName.ext` does a clean print of the current figure. The preferred strategy is to predefine options, so that `comgui('ImWrite')` alone is sufficient to generate a figure. This can be done by

- predefining properties in a `comgui PlotWd` call (including the file name as illustrated under `comgui ImFtitle`.
- or using the obsolete strategies of setting `ua.ImWrite` in the `iiplot PlotInfo` so that the proper data is used when a `curve` is displayed in `iiplot` or setting `ImWrite` in `comgui def.Legend` so that the proper configuration is used when a `def` is displayed in `feplot`.

`comgui('ImWrite',gf,RO)` with a figure handle given in `gf` and options stored in the `RO` structure, is the most general. `gf` can be omitted and will be taken to be `gcf`.

`RO` can be omitted if options are given as strings in the command. Thus `ImWrite-NoCrop` is the same as using `RO.NoCrop=1`.

For details for multi-image capture strategies (for example a set of modeshapes), see `iicom ImWrite`.

Acceptable options are detailed below.

- `.FileName` The default extension is `.png`. With no file name a dialog opens to select one. `RO.FileName` can be a cell array for a `ImFtitle` call.
- `.NoCrop=1` avoids the default behavior where white spaces are eliminated around bitmap images.
- `.FTitle=1` uses the title/legend information to generate a file name starting with the provided filename.

A typical example would be `comgui('imwrite-FTitle plots/root')` which will generate a `root_detail.png` file in local directory `plots`.

For a given plot, `comgui('imFTitle')` can be used to check the target name.

Using a cell `.FileName` calls `comgui ImFtitle` to let you build the file name from elements within the figure.

- `.LaTeX=1` displays LATEX commands to be used to include the figure in a file.
- `.objSet` provides an `comgui objSet` style. You can also combine predefined styles using a cell of the form `{'@OsDic(SDT Root)',{'fmt1','fmt2'}}`. The `'@ToFig'` can be used to clone the figure before printing to avoid modifying its appearance.
- `.clipboard` copies to clipboard.
- `.Java` To use screenshot strategies of the system current monitor display using java. This implies in particular that the content to capture is visible on screen when the command is used.
  - `.Java=1` uses java to do a screen capture of the figure content (undocked figure).
  - `.Java=2` captures the figure with the figure border (undocked figure). Use 2.2 to perform a clean crop arround the figure (if windows in your OS are surrounded by an unicolor rectangle)
  - `.Java=3` captures the dock containing the current figure.
  - `.Java=4` captures the content of the current tab in a tabbed pane without column headers.
  - `.Java=5` captures the pane containing the current tab (add the tab layout).
  - `.Java=6` captures the content of the current tab in a tabbed pane with column headers.
  - `.Java=7` captures the content of the tile containing the figure (figure + figure headers).
  - `.Java=8` captures the ExploTree of the UI.
- `.JavaT` To capture figures contents using java object methods (works for tables only)
- `.open=1` opens the image in a browser.
- `.Crop='all'` modifies the cropping option, see `comgui ImCrop`. Use `'no'` to avoid cropping.
- `.MultiExt={'.png','.fig'}` will allow saving of multiple versions of the same image.
- `.wobjSet` is used to insert the image into the current Microsoft Word file directly. `d_imw('get','WrW49C')` gives a sample format.

It is also possible to directly capture a graphical java object which contains getVisibleRect and getLocationOnScreen properties. Simply provide the java object as instead of a figure handle.

```
sdtweb sdt % Open sdt.html in the help browser
pause(2); % Wait for the display
desktop = com.mathworks.mde.desk.MLDesktop.getInstance;
r1=desktop.getGroupContainer('Help') % Get the java container of the help browser
% Save the HelpBrowser capture in the tempdir with name testjavacapture.png
comgui('imwrite testjavacapture',r1);
```

ImFtitle, ...

ImFtitle generates a file name for the figure based on current displayed content. Text is searched in objects with tags legend, ii_legend, in the axes title. By default all the text is concatenated and that can generate excessively long names so finer control is achieved by providing the FileName as a cell array in the comgui PlotWd call. The underlying mechanism to generate the string is described in comgui objString.

```
figure(1);clf; t=linspace(0,2*pi);h=plot(t,[1:3]'*sin(t));
legend('a','b','c');title('MyTit');
% Define target plot directory in the figure
cingui('objset',1,{'@PlotWd',sdtdef('tempdir')})

% Check name generation, from string
comgui('imftitle',1,{'@PlotWd','@title','.png'})
% Do a direct call with name building
comgui('imwrite',struct('FileName',{{'@PlotWd','@title','.png'}}))

% Predefine the figure save name in the userdata.Imwrite of current axis
comgui('PlotWd',1,'FileName', ...
   {'@Plotwd','@title', ... % Search for plotwd, use title name
   '@legend(1:2)','.png'}); % use first legend entry
comgui('imInfo') % See parameters
% check image name, display clickable link for image generation
comgui('imftitle')
sdtweb('_link','comgui(''Imwrite'')','Generate');

d_imw('Fn') % Standard names styles for tile name generation
```

dock

SDT uses some docking utilities that are not supported by MATLAB. The actual implementation is thus likely to undergo changes.

```
gf=11;figure(gf);clf; t=linspace(0,2*pi);h=plot(t,[1:3]'*sin(t));
figure(12);plot(rand(3));figure(13);mesh(peaks);
% set the dock name and position
comgui('objset',[11 12 13],{'@dock',{'name','MAC', ...
  'arrangement',[1 1 2;1 1 3], ... % Automated tile merging
  'position',[0 0 600 400],...
  'tileWidth',[.4 .6], ... % Fraction of columns
  'tileHeight',[.3 .7]}}); % Fraction of rows
pos=feval(iimouse('@getGroupPosition'),'MAC'); % group screen position
figure(14); % Add a new figure in specified tile
cingui('objset',14,{'@Dock',{'Name','MAC','Tile',11}});

feval(iimouse('@deleteGroup'),'MAC') % Delete group (and figures)
```

Capture of a dock group figure is possible with `comgui imwrite-Java3`

guifeplot,iiplot

```
cf=comgui('guifeplot -reset -project "SDT Root"',2);
comgui('iminfo',cf) % View what was set
```

Is used to force a clean open of an `feplot` figure. The option `-reset` is used to force emptying of the figure. The option `-project` is used to combine a call to `comgui PlotWd` to define the project.

Formatting styles `sdtroot OsDic` are also stored in the project.

objSet (handle formatting)

`cingui('objSet',h,Prop)` is the base SDT mechanism to generalize the MATLAB `set` command. It allows recursion into objects and on the fly replacement. `Prop` is a cell array of tag-value pairs classical in MATLAB handle properties with possible modifications. Three base mechanisms are object search, expansion and verification.

Object search `'@tag',value` applies property/values pairs stored in `value` to an object to determined on the fly. For example `'@xlabel'` applies to the xlabel of the current axis.

- `@xlabel` accepts a value that is a cell array that will be propagated for all x labels. A typical example would be `{'@xlabel',{'FontSize',12}}`. Other accepted components are `@ylabel`, `@zlabel`, `@title`, `@axes`, `@text`,
- `@axes`, `@figure` will search for parent or child axes objects
- `@tag` is assumed to search for object with the given tag, so that its properties can be set. For example `{'@ii_legend',{'FontSize',12}}` will set the fontsize of an object with tag `ii_legend`.
- `@tag(val)` allows the selection of a specific object by index when multiple objects with the same tag are found.
- `@ImFtitle` is used to store the cell array for image name generation see `comgui ImFtitle`. This must be set after displaying title and legend entries, since the information is stored in these objects.
- `@legend` generates the usual MATLAB legend
- `@ii_legend` allows a tick generation callback, see `ii_plp Legend`
- `@TickFcn` allows a tick generation callback, see `ii_plp TickFcn`
- `@ColorBar` allows handles properties of colorbar. This is illustrated under `fecom ColorBar`, but can be used for any figure.
- `@dock` handles docking operations, see `comgui dock`.
- `@ToFig` replicate the figure before applying operations. Property `{'cf',val}` can be used to force replication into figure `val` (use NaN for a new figure). Property `{'PostFcn',val}` can be used to allow execution of a callback after the figure replication. Property `{'leg',1}` uses the `iiplot ii_legend` object, while 2 transforms to a MATLAB legend.
- `@PlotInfo` calls `iicom PlotInfo` to initialize how data is displayed in an `feplot`/`iiplot` figure.

Expansion modifies the current property/value list by replacing a given entry.

- `'@OsDic(SDT Root)',{'val1','val2'}` seeks objset values in the `sdtroot OsDic`.
- `'','@tag'` is first expanded by inserting a series of tag-value pairs resulting from the replacement of `@tag`.

The two uses are illustrated below

```
% Define OsDic entries in project
sdtroot('SetOsDic',{'feplotA',{'Position',[NaN NaN 500 300]};
 'font12',{'@axes',{'fontsize',12},'@title',{'fontsize',12}}
 'grid',{'@axes',{'xgrid','on','ygrid','on','zgrid','on'}}
});
sdtroot('setOsDic', ... % Define a line sequence
   {'LiMarker',setlines(jet(5),{'-','--','-.'},'+ox*sdv^><ph')})
% Example of apply call
figure(1);plot(sin(linspace(0,4*pi)'*[1:3]))
cingui('objset',1,{'@OsDic(SDT Root)',{'feplotA','grid','LiMarker'}})
% Get OsDic data for given entry
sdtroot('cbosdicget',[],'ImLW75') % in project
cingui('fobjset','RepRef',{'@OsDic',{'feplotA','grid'}})
cingui('fobjset','RepRef',{'','@Rep{SmallWide}'})
```

Value replacement/verification performs checks/callbacks to determine the actual value to be used in the MATLAB set.

- `position` accepts `NaN` for reuse of current values. Thus `[NaN NaN 300 100]` only sets width and height.
- `@def` The value is a default stored in `sdt_table_generation('Command')`. One can search values by name within a cell array. This is in particular used for preset report formats `@Rep{SmallWide}` in `comgui ImWrite`.
- `xlim`, ... `clim` accept callbacks for the setting of limits. 'set(ga,"clim",[-1 1]*max(abs(get(ga,"clim"))))' is a typical example setting symmetric color limits.
- `'@setlines(''marker'')'` or `'@out=setlines(''marker'');'` are two variants where the value is obtained as the result of a callback. Note that the variant with `@out` must end with a semicolumn. This is illustrated in the example below.

```
figure(1);t=linspace(0,2*pi);h=plot(t,[1:3]'*sin(t));
cingui('objset',1, ...       % Handle to the object to modify
  {'','@Rep{SmallWide}', ... % Predefined figure type
   '@line','@setlines(''marker'')'}) % Line sequencing
cingui('fobjset','RepRef',{'','@Rep{SmallWide}'})
```

objString (string generation for title and file)

cingui('objString',h,SCell) is a mechanism to generate strings based on a set of properties. Elements of SCell are replaced when starting by an @, with implemented methods being

- @PlotWd is the base mechanism to find the plotting directory, see comgui PlotWd.

  @PlotWd/relpath is accepted in name generation to allow simple generation of relative paths.

- @tag(1:2) allows selection of a subset of objects when multiple exist. Typical are @legend(1) to select the first string of a MATLAB legend, or @ii_legend(1) for an SDT ii_plp Legend entry. @headsub for the text used by feplot to display titles.

- @colorbar seeks the string associated with a colorbar

- @cf.mdl.name or any variant based on @cf can be used to retrieve data in an *SDT* handle pointer.

This is used by comgui ImFtitle to generate figure names, but can also be used elsewhere (fe_range, ...). For example in title generation.

```
figure(1);clf;
t=linspace(0,2*pi);h=plot(t,[1:3]'*sin(t));title('MyTit')
legend('a','b','c');
SCell=   {'@Plotwd/plots', ... % Search for plotwd/plot
   '@title', ... % use title name
   '.png'}; % extension
cingui('objstring',1,SCell) % Handle of base object
```

ParamEdit

cingui('ParamEdit') calls are used to clarify filling of options data structures as detailed in section 7.17.3 .

def.Legend

The def.Legend field is used to control dynamic generation of text associated with a given display. It is stored using the classical form of property/value pairs stored in a cell array, whose access can be manual or more robustly done with sdsetprop.

Accepted properties any text property (see doc text) and the specific, case sensitive, properties

- set gives the initialization command in a string. This command if of the form 'legend -corner .01 .01 -reset' with
  - cornerx y gives the position of the legend corner with respect to the current axis.
  - -reset option deletes any legend existing in the current axis.
- string gives a cell array of string whose rows correspond to lines of the legend. $title is replaced by the string that would classically be displayed as label by feplot. Individual formatting of rows can be given as a cell array in the second column. For example {'\eta_1',{'interpreter','tex'}}.

```
[model,def]=hexa8('testeig');cf=feplot(model);
cf.data.root='\it MyCube';
def.Legend={'set','legend -corner .1 .9 -reset', ... % Init
 'string',{'$title';'@cf.data.root'}, ... % The legend strings
 'FontSize',12} % Other test properties
cf.def=def;
```

PlotWd

A key aspect of image generation is to define meta-data associated with a figure. These include, directory where the image will be saved, file name, possible inclusion in Word, PowerPoint, ... The Project tab defines the plot directory and possibly a file for inclusion. Other properties are set using the PlotWd command cingui('plotwd',gf,'@OsDic(SDT Root)') as illustrated below.

When initializing in a feplot figure, use cf=comgui('guifeplot -project "SDT Root"',2) to set the project information. Note that the older strategy using cf.def.PlotWd, cf.mdl.PlotWd are still used first if they exist. Similarly use cf=comgui('guiiiplot -project "SDT Root"',2) to set the project information of iiplot figures.

When refining formatting beyond specifying directory, insertion file, accepted property/value pairs (a structure can also be used but this is not the norm)

- '@OsDic(SDT Root)',list is used to extract property/values from the dictionnary. The (SDT Root) is the name of the figure from which dictionnary and project information is to be obtained from. The Project values is set.
- Project tag of project interface. Default would be SDT Root
- FileName cell array describing file name generation, see example in comgui ImFtitle. Note that the Fn.. OsDic entries allow generation of names from text present in the figure (labels, titles, ...).

- `objSet` cell array of objset commands to be performed before generating an image. This typically begins by a `@ToFig` to avoid modifying the original figure.
- `wobjSet` cell array of commands for insertion of the resulting image into MicroSoft Word, Powerpoint,Excel, LaTex, ... A sample entry is given by `d_imw('wrw49c')`.
- `'MultiExt',{'.png','.fig'}` cell array of extensions to save mutiple versions of given figure.

```
t=linspace(0,pi); % basic plot
gf=1;figure(gf);clf;plot(t,sin(t));
title('TestFigure');legend('a');
% Define the project directory
sdtroot('SetProject',struct('PlotWd',sdtdef('tempdir')))
% Prepare for image generation.
list={ ...  % List of OsDic entries, implemented in d_imw
  'ImToFigN', ... % Duplicate to new figure before ImWrite
  'FnTitle', ...  % Generate file name based on Title
  'WrW49C'        % Insert in word with 49% wide centered
  };
% Associate figure gf with project SDT Root
cingui('plotwd',gf,'@OsDic(SDT Root)',list)
comgui('iminfo',gf) % View what was set
comgui('imwrite',gf) % Actually insert image
```

A variety of predefined formats is available (and can be customized) with `OsDic`.

FitLabel

`comgui('fitlabel')` attempts to replace axes of the current figure so that `xlabel`, `ylabel`, ... are not cropped.

commode

Purpose

General purpose command parser for user interface command functions.

Syntax

```
Commode ('CommandFcn','ChainOfCommands')
```

Description

Commands and options are central to SDT. These strings are passed to functions to allow multiple variations in behavior. Accepted commands are listed in the help (text) and sdtweb (html) documentations (see iicom, fecom, feutil, etc.).

- commands are case insensitive, thus FindNode and findnode are equivalent. The uppercase is used to help reading.
- options can be separated by blanks : 'ch1' or 'ch 1' are the same.
- option values (that must be provided) are indicated *italic* in the HTML help and in brackets () in the text help.

  For example ch *i* indicates that the command ch expects an integer. ch 14 is valid, but ch or ch i are not.
- in the help alterative options are indicated by [c1,c2] (separated by commas).

  For example ch[,c] [*i*,+,-,+*i*,-*i*] means as a first alternative that ch and chc are possible. Then alternatives are *i* a number, + for next, - for previous, +*i* for shift by *i*. 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], fecom 'ch 14' or fecom('ch 1 4') but not fecom ch 1 4 where ch, 1 and 4 are interpreted by MATLAB as 3 separate strings.
- ; placed at the end of a command requests a silent operation as in MATLAB.
- When building complex commands you may need to compute the value used for an option. 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 chaining*: several commands separated by semi-columns `;`. The parsing is then done by `commode`.
- *scripting*: execute all commands in a file.
- *command mode*: replace the MATLAB prompt `>>` by a `CommandFcn>` which directly sends commands to the command function(s).

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.

```
Cam,string 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. Note that `istrue` is output in format double and not logical. See also `strncmp`.

```
Cam,string,format [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,ind [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.

```
-1 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.

`-3 date = comstr(CAM,[-3])`

Return the standard date string. Used by `ufwrite`, `naswrite`, etc. See also `date`, `datenum`.

`-4 CAM = comstr(CAM,[-4 nc ])`

Fills the string `CAM` with blanks up to `nc` characters.

`-5 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')`.

`-7 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.

`-17 Tab , comstr(tt,-17,'type')`

This is used to generate tabular output of the cell array `tt` to various supported types : `tab` (opens a java tab containing the table), `excel` (Microsoft Excel only available on windows), `html`, `csv` (comma separated values, readable by excel), `tex` (latex formatting), `text` printout to the command window.

```
% A sample table
tab=num2cell(reshape(1:10,[],2));tab(1,:)={'c1','c2'};
tname=nas2up('tempname o.html');
% RO option structure to format a table for HTML or java output
RO=struct('fmt',{{'%3i','%.1f'}}, ... % Formatting for each column
   'HasHead',1); % a header is provided as strings
RO.fopen={tname,'a+'};  % Opening information
RO.OpenOnExit=0;
RO.Legend=sprintf('<p>%s</p>','My HTML legend');
%  comstr(tab,-17,[],RO.fmt)
comstr(tab,-17,'html',RO);
sdtweb('_link',sprintf('web(''%s'')',tname))
% Show the table in JAVA tab
```

```
comstr(tab,-17,'tab',RO);
% Generate tex output of java tabs
comstr(struct('FigTag','SDT Root'),-17,'tex');
comstr(gcf,-17,'tex');
```

Accepted fields for the options structure are

- `.fmt` cell array of column formatting instructions. These can be strings `%.1f,%i,%.2g` which are passed to `sprintf`. They can also be java strings `java.lang.String('0.00%')` which are then parsed using `java.text.DecimalFormat`.
- `.ColumnName` cell array with first row giving column names. `.ColumnName(:,3)` can also be used to store the column format. `.ColumnName(:,4)` can also be used to store cell coloring data, see section 7.18 .
- `.HasHead` if non zero, skips lines of strings

Fields specific for HTML generation are

- `.name` is used to define a title for the table.
- `.fopen` used for HTML generation. For example `{tname,'a+'};` is for append. `.OpenOnExit` asks to open the file in the web browser.

Fields specific for JAVA tabs are

- `.setSort` activates row sorting in java tables. 1 : basic sort, 2: selectable sort. 3 : tree table.
- `.name` is used to define a tab name.
- `.FigTag` tag or handle for figure where the tab should be displayed.
- `.ColWidth` vector of column width in pixels.
- `.groupable` used with `.setSort=3` to specify columns that will be used to generate the tree.
- `jProp` accepts `tag,value` pairs. `'ResizeMode','Off'` to fix colums for example. `'MousePressed',data` gives a cell array used to store events that should be handled by the table (see `menu_generation('jpropcontext',ua,'Tab.ExportTable')`).
- `.ColumnName` second row can give alignment `'right'`. Third row can give column formatting (alternatively, the `.RowFmt` can be used). Row 4 can be used to define a color based on a `CritFcn`.

-38 [i0,st2]=comstr(st1,-38)

Checks whether provided string `st1` is valid to be a structure field. Output `i0` is a boolean, true if valid, false otherwise. Output `st2` is equal to input `st1` if the string is valid. If not, `st2` is an

alternative valid suggestion based on st1.

See also

commode

curvemodel

Purpose

Handle object for implicit representation of curves.

Syntax

```
h=curvemodel('Source',r1,'yRef',fun,'getXFcn',{fun,fun,fun}, ...
   'DimPos',[1 3 2]);
```

Description

Multi-dim curve are multi-dimensional arrays (.Y field) with information about the various dimensions (.X,.Xlab fields). curvemodel store similar data sets but provide methods to generate the .X,.Xlab,.Y fields content dynamically from an information source.

curvemodel objects are derived from MATLAB handle objects. If you copy an object's handle, MATLAB copies only the handle and both the original and copy refer to the same object data.

The principle of curve models is that the computation only occurs when the user seeks the required data.

Important fields are

- .Source contains the data to be used as source. The source can be a pointer. For example cf.v_handle.Stack{'def1'} can be used to point to a set of deformations stored in a feplot, or iiplot stack.
- .DimPos is used to allow permutations of the array dimensions (implicit equivalent of permute(c.Y,c.DimPos).
- .xRef is a cell array of length the number of dimensions in .Y allowing the extraction from the source.

Documented methods are

- .GetData : creates a copy of the full implicit data.

This functionality mostly undocumented. Support functions are process_r that handles delayed signal processing requests, ii_signal that supports curvemodel commands associated with signal processing. The following is an example for users willing to dig into the code.

```
C1=d_signal('RespsweepSpec') % Create a spectrogram model
C2=C1.GetData; % create a copy where the spectrogram is computed
C2.PlotInfo=ii_plp('plotinfo 2D');
iicom('curveinit','Spectro',C2);
```

db, phaseb

Purpose

Compute the decibel magnitude.
Compute the unwrapped phase in degrees.phase

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`

ex2sdt

Purpose

Interface between EXCITE and SDT (part of FEMLink).

Syntax

```
ex2sdt('read',FileName);
ex2sdt('post');
```

read[*.cff, *.gid]

```
ex2sdt('Read','fname.cff') % Read .cff file and display in feplot
```

This command can be used to read some Excite specific output files :

- `.cff` file can be used to export model geometry. Model is read and displayed in `feplot`.
- `.gid` file can be used to export time curve at a current DOF. A full directory can be read : `ex2sdt('Read','Directory.gid')`. Curves are displayed in `iiplot`.

ConvertAsMat

`ex2sdt('ConvertAsMat')` This command aims to convert all Excite results of a given directory as SDT mat files (typically `RO mdl` and `def` variables) that can be explorated and post treated through the `ex2sdt UIScan` command.

```
sdtroot('SetProject',struct('ProjectWd','projectpath','root','resultroot'))
ex2sdt('Post')
```

First a root project must be opened, defining at least :

- ProjectWd : the main project directory that contains the results of the time simulation.
- root : the root of the filenames where model and results are stored.

The result folder must contains

- the model file, and if needed the associated reduction basis file. The model file should be (in preference order):

  - a root.OUT2 nastran output2 file from the DMAP condensation step.
  - a root.NAS nastran bulk file.
  - a root.cff excite file (to be implemented).

- if needed, the file that contains the restitution matrix in the case of meshed part reduced using the AVL DMAP. This file is read and lead to a model.TR reduction basis, that can be used to expand the displacement form the reduced model to the full displacement field (and so animate the time deformation in feplot).
  - a matlab file, root_X2OA.mat, that is obtained by the ex2sdt TextOp4 command, that converts the original export text OUT4 file to a Matlab binary file that is more convenient to use (it can be read as an HDF handle to save memory).
  - directly the original root_X2OA.OUT4 export, which is a text file. This case is only suitable for files that are lower than 300 MB.

- as many subfolders as simulation results. For the moment each simulation typically corresponds to a specific rotation speed (so each subfolder name should end by the rotation velocity in RPM, for example study.2000 for the 2000 RPM speed case) : this will be generalized to obtain simulation information and build a simulation parameter data structure RO, in order to perform dirscan in the generic SDT process (see sdtweb fe_range) (XXX need to find way to recover those parameters from Excite ouputs...). The result files should be, by order of preferences:
  - a root_SOL109.INP4 nastran input4 text file that contains displacements, velocities and acceleration at each (reduced) DOF. This is the more compact and usable output. Corresponding time and angle are then read in the util_batch_list.out log file.
  - a number of root-NodeID-DOFID-REL.GID files, each one containing the displacement, velocity and acceleration in a given of the 6 directions at a node of the model. Some developpements are needed to use this strategy (INP4 should be prefered), that is beside very time-consumming.

  The input parameters can be get from 2 different files:
  - summary.xml : that can be read with RO=feval(ex2sdt('@parseXML'),'summary.xml'). Developpement must be done.
  - simulation_report.out : a text log file that can be read with RO=feval(ex2sdt('@readReportOut'),'simulation_report.out').

UIScan

`ex2sdt('UIScan')` This command can be used to scan a given directory (defined in the Project tab) and then perform some post-treatment (compute campbell diagrams and animate displacements or velocities as color map,...) and explorate data through UI.

TextOp4

This command is experimental.
`ex2sdt('TextOp4','filename_X20A.OUT4')`
It aims to convert an ascii X20A.OUT4 file, to a binary HDF .mat file. This operation is bufferized so that the 2 GB memory limitation of old 32 bits Matlab can be bypassed.

See also

FEMLink

fe2ss

Purpose

Build state-space or normal mode form from FE model.

Syntax

```
[sys,TR] = fe2ss('command [options]',MODEL)
[sys,TR] = fe2ss('command [options]',MODEL,C)
[nor,TR] = fe2ss('command -nor', ...)
TR       = fe2ss('command -basis', ...)
```

Description

`fe2ss` is meant to allow users to build state-space (see section 5.4) and normal mode models from full order model matrices. Accepted commands are detailed below. Accepted command options

- `-nor` outputs the normal mode model data structure (see section 5.2).
- `-basis` outputs the reduction basis is the structure `TR`
- `-se` outputs a reduced superelement
- `-loss2c` performs estimates viscous damping based on hysteretic models
- `-cpx 1` computes complex modes and uses a call to `res2ss` to compute the state space model. `-cpx 2` uses first order correction in the `fe_ceig` call before using `res2ss` to build the state-space model. This is currently only available for a `Free command`.
- `-dterm` includes static correction as a D term rather than additional modes. The associated full order shapes are stored in `TR.bset`.
- `-ind` specifies indices of modes to be kept. Others are included as a D term.

The procedure is always decomposed in the following steps

- call `fe_reduc` build a reduction basis given in `TR.def` (see section 6.2). This usually includes a call to `fe_eig` with options *`EigOpt`* provided in the `fe2ss` command
- call `fe_norm` to orthonormalize the basis with respect to mass and stiffness (obtain a model in the normal mode form (5.4), see section 5.2) and eliminate collinear vectors if any
- call `nor2ss` or project model matrices depending on the number of outputs

The TR output argument, contains the modeshapes followed by residual vectors, is given so that the user can display modeshapes in feplot with cf.def=TR or call nor2ss repeatedly without computing the basis again. The later is in particular useful for changes in the sensor configuration which have no effect on the retained basis. -nor and -basis can be used to generate the corresponding outputs.

High level input arguments are a MODEL (see section 4.5) with a case defined in the model which **must** contain load and sensor entries (see fe_case).

Damping can be specified multiple ways.

- modal damping of viscous form can be given in the model (using a DefaultZeta case entry as shown below) or as an additional argument C which can be a system damping matrix, a scalar uniform damping ratio or a vector of damping ratios.
- defining modal damping using an inline function. For example to set 1% below 3000 Hz and 5% above use

```
model=stack_set(model,'info','DefaultZeta', ...
    @(f)double(f/2/pi<3000)*.01+double(f/2/pi>=3000)*.05);
```

- using material loss factors and adding the -loss2c option described above.

in the model (using a DefaultZeta case entry for example), or given as an additional argument C which can be a system damping matrix, a scalar uniform damping ratio or a vector of damping ratios.

The following example compares various damping models.

```
mdl=demosdt('demo ubeam mix');cf=feplot;
mdl=fe_case(mdl,'SensDof','Out',[343.01 343.02 347.03]', ...
    'FixDof','base','z==0')
freq=linspace(10,1e3,2500)';mdl=stack_set(mdl,'info','Freq',freq);
% uniform 1 % modal damping
mdl=stack_rm(mdl,'info','RayLeigh');
mdl=stack_set(mdl,'info','DefaultZeta',.01);
[sys,T] = fe2ss('free 6 10',mdl);
qbode(sys,freq*2*pi,'iiplot "Modal"');
% Rayleigh damping with 1 % viscous at 200 Hz, see sdtweb('damp')
mdl=stack_rm(mdl,'info','DefaultZeta');
mdl=stack_set(mdl,'info','Rayleigh',[0 .01*2/(200*2*pi)]);
[sys2,T] = fe2ss('free 6 10',mdl);
qbode(sys2,freq*2*pi,'iiplot "Rayleigh"');
% Estimate viscous from hysteretic damping
[sys3,T] = fe2ss('free 6 10 -loss2c',mdl);
```

```
qbode(sys3,freq*2*pi,'iiplot "Loss"');

iicom('iix',{'Modal','Rayleigh','Loss'});

% display full response
RB=struct('f',cf.Stack{'Freq'},'u',eye(5,1))
cf.def=fe2ss('sysdef',sys,T,RB);
% use iimouse('cursorOnFeplot') to see deformations at various freq.
```

SysDef

The command is used to generate a restitution of a forced response on all DOF in TR. The calling format is fe2ss('sysdef',sys,TR,RB) with fields of the option structure being

- .f frequency in Hz. or .w frequency in rad/s.
- .u input possibly a vector that should be consistent with sys.b.

Free [, Float] [, -dterm] *EigOpt*

See fe_reduc Free for calling details, this generates the classical basis with free modes and static correction to the loads defined in the model case (see fe_case). With the -dterm option, the static correction is given as a D term rather than additional modes.

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. In the example below, note the high accuracy up to 200 Hz.

```
mdl=demosdt('demo ubeam');cf=feplot;
mdl=fe_case(mdl,'SensDof','Out',[343.01 343.02 347.03]', ...
    'FixDof','Base','z==0')
freq=linspace(10,400,2500)';mdl=stack_set(mdl,'info','Freq',freq);
% uniform 1 % modal damping
mdl=stack_rm(mdl,'info','RayLeigh');
mdl=stack_set(mdl,'info','DefaultZeta',.01);
```

```
[sys,T] = fe2ss('CraigBampton 5 10', ...
  fe_case(mdl,'DofSet','IN',314.01));
qbode(sys,freq*2*pi,'iiplot "Craig"');

% Same with free modes
[sys2,T2] = fe2ss('Free 5 10', ...
  fe_case(mdl,'Remove','IN','DofLoad','IN',314.01));
qbode(sys2,freq*2*pi,'iiplot "Free" -po');

iicom('iixOnly',{'Craig','Free'});iicom(';sub 1 1;ylog')
```

Low level input format

The obsolete low level input arguments are those of fe_reduc with the additional damping and output shape matrix information.

```
[sys,TR] = fe2ss('command',m,k,mdof,b,rdof,C,c)
```

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 5.1). Must be coherent with mdof.

Standard bases used for this purpose are available through the following commands.

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(cf,'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 4.4 . 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).

- `cf1=feplot` returns a pointer to the current `feplot` figure (see section 4.4.3). 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)`. If many `feplot` figures are open, one can define the target giving an `feplot` figure handle `cf` as a first argument.
- 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

`AddNode,Line`

These commands start to implement direct model modification in the feplot figure. Sample calls are illustrated in section 2.7.1 .

`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 `AnimStart` 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 can also use `fecom('AnimTime` *StepInc*`')` to define the step increment of the animation. You may need to fix the color limits manually using `cf.ua.clim=[0 1e3]`.

```
demosdt('demobartime'); fecom AnimeTime5;
```

Accepted `Anim` options are

- `Freq` 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`.
- `Time` starts the animation in a mode that increments deformations while preserving the amplification. This is appropriate for animation of time responses.
- `One` animates the current axis only rather than the default (all).
- `Col` sets color animation to dual sided (alternates between a max value and its opposite) rather than the default of no animation. You can animate colors without deformations if you define colors for the current selection without defining a deformation.
- `Slider` *`On,Off,Tog`* opens an slider to select deformation.

.

Animation speed is very dependent on the figure renderer. See the `fecom Renderer` command.

`AnimMovie` *`step`*

SDT supports creation of movies using `VideoWriter, imwrite, avifile`.

Command option -crop calls comgui ImCrop to crop borders, ... You can use the .Movie field in iicom ImWrite to generate multiple files.

Typical uses are illustrated below

```
cf=demosdt('DemoGartfePlot'); fecom('ColordataEvalZ-edgeAlpha.1');% Load an example
fecom('MovieProfiles') % List profiles (supported file types)

tname=nas2up('tempname.gif');
R1=fecom('AnimMovie',tname) % Base give a name
R2=fecom('AnimMovie-CropEqual',tname) % ask to crop all white

% More advanced specify properties and shapes
R2=struct('FileName',{{sdtdef('tempdir'),'Gart','@ii_legend','.gif'}}, ...
 'prop',{{'Quality',100,'FrameRate',10}}, ... % VideoWriter properties
 'CropFcn',{{'comgui','imCropEqual'}}, ... % Do cropping
 'PostFcn','camorbit(5,0)'); % Callback after each step
% R2=fecom('AnimMovie 10',R2); % Here save 10 animation steps
R2=fecom('ImWrite',struct('ch',7:8,'Movie',R2)); % Generate two movies

% Use a Matlab Movie
R3=struct('Profile',{{'','Matlab','movie'}});
R3=fecom('AnimMovie 10',R3); % Get a Matlab Movie in R3.M
```

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*] [,-alpha*i*]

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. See also fecom ColorBar and fecom ColorLegend commands.

Accepted options for the command are

- -alpha val can be used to set face transparency. This is only valid using OpenGL rendering and is not compatible with the display of masses (due to a MATLAB rendering bug).
- -edgealpha val is used for edge transparency
- -ColorBarTitle "val" is used to open a colorbar with the appropriate title (see ColorBar and ColorScale commands). A .ColorBar field can be used for calls with a data structure input.

Accepted ColorData commands are listed below

Eval	fecom('ColorData EvalZ') does dynamic evaluation of the color field based on current displacements. Accepted eval options are x,y, z, a for single axis translations or translation amplitudes. RadZ,TanZ for radial and tangential displacement (assumed cylindrical coordinates with z axis).
Ener	the preferred method is now to compute energies and display using ColorDataElt as detailed in fe_stress feplot. The old command fecom('ColorData EnerK') is considered obsolete.
Group, Mat, Pro, i	fecom('ColorDataGroup') defines a color for each element group, Mat for each MatId, and Pro for each ProId. ColorDataI gives a color for each separate triplet. A color map can be given as a second argument. ColorData *Group* -edge affects colors to nodes rather than surfaces and displays a colored wire-frame. fecom('ColorMatId',[100 0 0 1]) lets you control colors associated with materials by setting RGB color value associated to MatId=100 in the info,MatColor case entry. Similar behavior is obtained for ColorProId and ColorGroupId The color animation mode is set to ScaleColorOne.
Stress	the ColordataStress$i$ command defines the selection color by calling fe_stress with command Stress$i$. The color animation mode is set to ScaleColorOne. This requires material and element properties to be defined with InitModel.
x, y, z, all,*DOF*	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.
Uniform	in this mode the deformation/object index is used to define a uniform color following the axis ColorOrder.
Elt	fecom('ColorDataElt',data) specifies element colors. Nominal format is a curve (see fe_stress Ener and fe_stress feplot) or a struct with .data .EltId. Older formats are a struct with fields .data .IndInElt or two arguments data,IndInElt.
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* for each deformation (these are assumed to be consistent with the current deformation set). Values are scaled, see the ScaleColor command. 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.

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 ..., 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('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.

ColorBar, ColorMap

`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. Editing of display is done with additional arguments `fecom('colorbar','CustomField',NewVal,...)`, where `CustomField` is a standard `colorbar` field, and `NewVal` the custom value to set. See `comgui objSet` for details on this generic SDT procedure.

`fecom ColorBarOff` is used to reinitialize a subplot without a color bar.

`fecom('colorMap')` calls `ii_plp('ColormapBand')` to generate specialized color maps. See `ii_plp ColorMap` for details.

In the following example, one plots the actual z displacement using a custom colorbar.

```
cf=demosdt('DemoGartfePlot');
fecom('colordataEvalZ -edgealpha .1')
% Disp in CM (*100), 2sided ([-cmax cmax]), instant (updated scale)
fecom('ColorScale Unit 100 2Sided Instant');
fecom('colorbar',d_imw('get','CbTR','String','z [mm]'));
% sdtweb d_imw('cbTr') % To see code of typical colorbar styles
```

```
fecom('colormapjet(9)');
```

A .ColorBar field can be used for ColorData calls with a data structure input.

ColorAlpha

fecom ColorAlpha starts a specific coloring mode where the transparency is indexed on the colormap level. This can be used to highlight high strain areas in volume models. -EdgeAlpha val may be used to make the edges transparent.

Uniform transparency of faces and edges is obtained using the FaceEdgeAlpha entry in the object context menu or with a command of the form below.

```
d_ubeam; cf=feplot;
% Use Value based alpha and Set the edges to be 10% transparent
fecom('ColorAlpha -edgealpha .1');
```

ColorScale

Once colors defined with fecom ColorData, multiple scaling modes are supported. fecom('ColorScale') displays current mode. For calling examples, see fecom ColorBar. The modes are accessible through the feplot:Anim menu.

- Tight corresponds to a value of [cmin cmax]. cf.ua.clim can be used to force values.
- 1Sided corresponds to a value of [0 cmax]. This is typically used for energy display.
- 2Sided corresponds to a value of [-cmax cmax]. This is typically used for translations, stresses, ...
- Fixed the color limits set in cf.ua.clim are used.

- Off the values are set at during manual refreshes (calls to fecom('ch') but not during animation. This mode is useful to limit computation costs but the color may get updated at the end of an animation.
- Instant the values of cmin,cmax are obtained using the current deformation.
- Transient the values are obtained using a range of deformations. For time domain animation, estimation is done dynamically, so that you may have to run your animation cycle once to find

the true limit.

- One does not scale color deformations (default starting with SDT 6.4)
- Unit *coef* defines a fixed color scaling coefficient. This is typically used to provide more convenient units (1e-6 to have stress colors in MPa rather than Pa for example).
- Def uses the amplification coefficient set for the associated deformation.

Cursor

If a time deformation is defined in the feplot figure, one can see time curve at a specific node using fecom CursorNodeIiplot command. A node cursor then appears on the feplot displayed model, and clicking on a node shows corresponding curve in the iiplot figure. Reciprocally one can show a cursor on the iiplot curve to show corresponding time deformation in feplot using iicom CursorOnFeplot command. Note that this functionality should only be used for small models.

Following example let you test this functionality.

```
model=femesh('testhexa8'); cf=feplot(model); model=cf.mdl; % simple cube
data=struct('def',[1 1 1 1]','DOF',[5 6 7 8]'+.03,...
    'curve',fe_curve('test sin 10e-2 5000 1 5000e-4'));
model=fe_case(model,'DofLoad','topload',data); % sin load
model=fe_case(model,'FixDof','basefix','z==0'); % fix base
model=fe_time('timeopt newmark .25 .5 0 1e-4 5000',model); % time computation
cf.def=fe_time(model); % show time animation

fecom CursorNodeIiplot % display cursor on feplot
ci=iiplot;iicom(ci,'ch',{'NodeId',5}) % Test the callback

iicom CursorOnFeplot   %  display cursor on iiplot

% Cursor following animation
fecom(sprintf('AnimCursor%i Start100',ci.opt(1)))
```

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.

LabFcn

Titles for each deformation should be generated dynamically with the def.LabFcn callback. def=fe_def('lab',def) attempts to provide a meaningful default callback for the data present in the def structure.

The callback string is interpreted with a call to eval and should return a string defining the label for each channel. Local variables for the callback are ch (number of the channel currently displayed in feplot) and def (current deformation).

For example def.LabFcn='sprintf(''t=%.2f ms'',def.data(ch)*1000)' can be used to display times of a transient response in ms.

fecom('TitOpt111') turns automatic titles on (see iicom). fecom('TitOpt0') turns them off.

Legend, Head, ImWrite

Placing a simple title over the deformation can be to coarse. Defining a comgui def.Legend field provides a more elaborate mechanism to dynamic generation of multi-line legends and file name (to be used in iicom ImWrite).

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)=def`. Where `cf` a *SDT* handle to the figure of interest and i the deformation set you which to modify (if only one is defined, `cf.def` is sufficient). Acceptable forms to specify the deformation are

- `def` is a structure with fields `.def`, `.DOF`, `.data`. Note that `.Legend` and `.LabFcn` can be used to control associated titles, see `comgui def.Legend`.
- `{mode,mdof,data}` a set of vectors, a vector of DOFs. For animation of test results, `mdof` can be given using the 5 column format used to define arbitrary sensor directions in `fe_sens`. The optional `data` is a vector giving the meaning of each column in `mode`. `fecom head` is used to generate the label.
- `ci.Stack{'IdMain'}`, see section 2.4 for identification procedures and section 5.6 for the pole residue format
- `[]` resets deformations
- `{def,'sensors'}` defines sensor motion in a case where sensors are defined in the case (that can be accessed through `cf.CStack{'sensors'}`). It is then expected that `def.DOF` matches the length of the sensor `tdof` field).
- `{def,TR}` supports automatic expansion/restitution, see illustrated in the `fe_sens WireExp` command. The same result can be obtained by defining a `def.TR` field.

`feplot(cf,'InitDef',data)` is an alternate calling format that defines the current deformation. `InitDef` updates all axes. `InitDefBack` returns without updating plots.

load, 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.

Lower level calls 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). Note that cf.model=UFS(1) for a data stack resulting from ufread and cf.model=Up for type 3 superelement.

Load from file fecom('Load','FileName') will load the model from a binary FileName.mat file. By default the variable model is searched in the file. fecom('FileImportInfo') lists supported import formats.

The following variables are looked for in the .mat file

- model a model structure.
- def a def structure that will be loaded by default in cf.def
- cf_sel*i*, with *i* a number, a sel selection structure that will be loaded and stored in cf.sel(i).

The following command options apply to command load for specific applications

- -back is used to load, but **not display** the model (this is used for very large model reading).
- -Hdf loads a model from a HDF5 .mat file but retains most data at v_handle pointers to the file.
- -sLin loads a model and generates a display using cf.sel='-linface'. This is needed for larger models.
- -noDef skips loading deformation curves when present.
- -skipFSE skips HDF loading of external data stored in model.fileSE

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 3.1).

`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.

Pro

`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 4.4 .

`feplot('ProViewOn')` turns entry viewing on.

Renderer[Opengl,zBuffer,Painters][,default]

This command can be used to switch 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.

`Save, FileExport`

Save the model to a `.mat` file or export it to supported formats.
`fecom('FileExportInfo')` lists supported export formats.

`fecom('Save -savesel file.mat'` also saves the selection(s) which allows faster reload of large models. `fecom('Save -savedef file.mat'` also saves the deformations(s).

`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. 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, and amplitude of other objects 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.

ScaleDof*i* 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.

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 (fecom ColorData in particular), use a call of the form cf.sel(i)={*'SelectionString'*,*'OtherProp'*,*OtherPropData*}.

To return to the default selection use fecom('SelReset').

fecom('Sel ... -linface') can be used to generate first order faces for second order elements,

which allows faster animation.

Callbacks to customized selections is also available. One can then provide a selection starting with `@`, the output will be evaluatued on-the-fly. The function must rethrow in order `i1`, `el0` and `i2` as

- `i1` the indices of the selected elements in `cf.mdl`.
- `el0` the elements selected in `cf.mdl`. This can be the result of a transformation, *e.g.* face elements from a `selface` based selection.
- `i1`

the indices of the selected elements in `cf.mdl`, including the element header rows. The function is called as `[i1,el0,i2]=eval(CAM(2:end));`

`SetObjectcf.o(1)= ...  fecomSetObjset` *i* `[,ty` *j*`] ...`

Set properties of object ***i***. Plots generated by `feplot` are composed of a number of objects with basic properties

- `ty` 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).
- def ***k*** index of the deformation set, stored in `cf.def(i)`, see`fecom InitDef`.
- ch ***k*** channel (column of deformation)
- sel ***k*** index of display selection. See `fecom Sel`.
- scc ***k*** scaling coefficient for the deformation.

The following example illustrates how the `SetObject` can be used to create new objects or edit properties of existing ones.

```
 cf=feplot(femesh('testquad4 divide 2 2'));
 cf.sel(2)='withnode {x==0}';
% Display objects in current axis
 cf
% Copy and edit one of the object lines to modify properties
 cf.o(1)='sel 1 def 1 ch 0 ty1'; % make type 1 (surface)
% Set other MATLAB patch properties
 cf.o(1)={'sel 2 def 1 ch 0 ty1','marker','o'}
```

```
% Multiple object set, object index is row in cell array
 fecom(cf,'setobject',{'ty1 sel 2 ty','ty2 sel 1'})
% remove second object by empty string
 cf.o(2)=''
```

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.
Fi...	The sdtroot OsDic utilities are now used to allow customization of plot initialization. d_imw('Fi') lists predefined init sequencees.
Bas	shows triaxes centered a the position of each local basis. The length of the triax arrow is specified by option -deflen*len*. Option DID places the origin of each triax at a node using this displacement frame.
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 plot 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). Nota : to see the real orientation, use the fecom('scaleone'); instruction. fecom('ShowUndef',MAP) also displays the underlying structure. MAP can also be a stack entry containing orientation information (see pro.MAP) or an element selection, as in the example below demosdt('demogartfeplot');fecom('ShowMap','EltName quad4')
NodeMark	fecom('shownodemark',1:10,'color','r','marker','o') displays the node positions of given NodeId (here 1 to 10) as a line. Here a series of red points with a o marker. You can also display positions with fecom('shownodemark',[x y z],'marker','x'). Command option -noclear allows to overlay several shownodemark plots, *e.g.* to show distinct sets of nodes with different colors at once. This can also be obtained by providing a cell array of node numbers.
Traj	fecom('ShowTraj',(1:10)') displays the trajectories of the node of NodeIds 1 to 10 for current deformation. Command option -axis is used to display axis node trajectories.
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').
DockXYZ	generates a dock with 3 subplots showing colors in the x, y and z directions.

Once the basic plot created, you can add other objects or modify the current list using the `Text` 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)`. Text properties can be given as extract arguments, for example `fecom('textnode',i,'FontSize',12,'Color','r')`. One can customize specific text display attached to nodal positions by directly providing a structure with fields `.vert0`, a 3 column matrix of nodal positions (that can be independent from the mesh) and `.Node` a 1 column cell array with as many lines as `.vert0` containing strings to be displayed.

`TextDOF` displays the sensor node and direction for the current sensor.

`TextDOF Name` displays sensor labels of a `cf.CStack{'Name'} SenDof` entry. Additional arguments can be used to modify the text properties. `fecom('textdof')` displays text linked to currently declared sensors, see `feplot InitSens` command (note that this command is being replaced by the use of `SensDof` entries).

`TextMatId` places a label in the middle of each material area. `TextProId` does the same for properties.

`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, None]`

Undeformed structure appearance. The undeformed structure is shown as a line which is made visible/invisible using `UnDef` (`UnDefNone` forces an invisible mesh). 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`. `iimouse('viewclone',[cf.opt(1) cg.opt(1)])` can be used to link animation and orientation of two `feplot` figures. This is in particular used in `ii_mac`.

See also

`feplot`, `fe_exp`, `feutil`

femesh

Purpose

Finite element mesh handling utilities.

Syntax

```
femesh CommandString
femesh('CommandString')
[out,out1] = femesh('CommandString',in1,in2)
```

Description

You should use `feutil` function that provides equivalent commands to `femesh` but using model data structure.

`femesh` provides a number of tools for mesh creation and manipulation. `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('Reset');
femesh('Testquad4');                    % one quad4 created
femesh('Divide',[0 .1 .2 1],[0 .3 1]);  % divisions
FEel0=FEel0(1:end-1,:);                 % suppress 1 element in FEel0
femesh('AddSel');                       % add FEel0 into FEelt
FEel1=[Inf abs('tria3');9 10 12  1 1 0];% create FEel1
femesh('Add FEelt FEel1');              % add FEel1 into FEelt
femesh PlotElt                          % plot FEelt
```

AddNode [,New] [, From i] [,epsl *val*]

Combine, append (without/with new) FEn0 to FEnode. Additional uses of AddNode are provided using the format

[AllNode,ind]=femesh('AddNode',OldNode,NewNode);

which combines NewNode *to* OldNode. AddNode finds nodes in NewNode that coincide with nodes in OldNode and appends other nodes to form AllNode. ind gives the indices of the NewNode nodes in the AllNode matrix.

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]=femesh('AddNode From 10000',OldNode,NewNode); gives node numbers starting at 10000 for nodes in NewNode that are not in OldNode.

SDT uses an optimized algorithm available in feutilb. See feutil AddNode for more details.

AddTest [,-EGID *i*][,*NodeShift*,Merge,Combine]

Combine test and analysis models. When combining test and analysis models you typically want to overlay a detailed finite element mesh with a coarse wire-frame representation of the test configuration. These models coming from different origins you will want combine the two models in FEelt.

By default the node sets are considered to be disjoint. New nodes are added starting from max(FEnode(:,1))+1 or from *NodeShift*+1 if the argument is specified. Thus femesh('addtest'',TNode,TElt) adds test nodes TNode to FEnode while adding NodeShift to their initial identification number. The same NodeShift is added to node numbers in TElt which is appended to FEelt. TElt can be a wire frame matrix read with ufread.

With merge it is assumed that some nodes are common but their numbering is not coherent. femesh('addtest merge',NewNode,NewElt) can also be used to merge to FEM models. Non coincident nodes (as defined by the AddNode command) are added to FEnode and NewElt is renumbered according to the new FEnode. Merge-Edge is used to force mid-side nodes to be common if the end nodes are.

With combine it is assumed that some nodes are common and their numbering is coherent. Nodes with new NodeId values are added to FEnode while common NodeId values are assumed to be located at the same positions.

You can specify an EGID value for the elements that are added using AddTest -EGID -1. In particular negative EGID values are display groups so that they will be ignored in model assembly operations.

The combined models can then be used to create the test/analysis correlation using fe_sens. An application is given in the gartte demo, where a procedure to match initially different test and FE coordinate frames is outlined.

Divide *div1 div2 div3*

Mesh refinement by division of elements. Divide applies to all groups in FEel0.

See equivalent feutil Divide command.

```
% Example 1 : beam1
femesh('Reset');
femesh(';Testbeam1;Divide 3;PlotEl0'); % divide by 3
fecom TextNode

% Example 2 : you may create a command string
number=3;
st=sprintf(';Testbeam1;Divide %f;PlotEl0',number);
femesh('Reset');
femesh(st);
fecom TextNode

% Example 3 : you may use uneven division
```

```
femesh('Reset');femesh('testquad4'); % one quad4 created
femesh('DivideElt',[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.

```
femesh('Reset');femesh('testquad4'); % one quad4 created
femesh('RepeatSel 2 0 0 1'); % 2 quad4 in the same group
femesh('DivideInGroups');    % 2 quad4 in 2 groups
```

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).

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*.

You can create irregular extrusion giving a second argument (positions of the sections for an axis such that tx ty tz is the unit vector).

See feutil Extrude for more details.

```
% Example 1 : beam
femesh('Reset');
femesh('Testbeam1'); % one beam1 created
femesh(';Extrude 2 1 0 0;PlotEl0'); % 2 extrusions in x direction

% Example 2 : you may create the command string
number=2;step=[1 0 0];
st=sprintf(';Testbeam1;Extrude %f %f %f %f',[number step]);
femesh('Reset');
femesh(st);  femesh PlotEl0
```

```
% Example 3 : you may use uneven extrusions in z direction
femesh('Reset'); femesh('Testquad4')
femesh('Extrude 0 0 0 1', [0 .1 .2 .5 1]); %
% 0 0 0 1        :  1 extrusion in z direction
% [0 .1 .2 .5 1] :  where extrusions are made
femesh PlotEl0
```

FindElt *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` (optional) the associated element description matrix. `FindEl0` applies to elements in `FEel0`.

When operators are accepted, equality and inequality operators can be used. Thus `group~=[3 7]` or `pro < 5` are acceptable commands. See also `SelElt`, `RemoveElt` and `DivideGroup`, the `gartfe` demo, `fecom` selections.

FindNode *Selectors*

Find node numbers based on a number of selectors listed in section 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.

As an example you can show node numbers on the right half of the `z==0` plane using the commands

```
fecom('TextNode',femesh('FindNode z==0 & x>0'))
```

Following example puts markers on selected nodes

```
model=demosdt('demo ubeam'); femesh(model); % load U-Beam model
fecom('ShowNodeMark',femesh('FindNode z>1.25'),'color','r')
fecom('ShowNodeMark',femesh('FindNode x>0.2*z|x<-0.2*z'),...
      'color','g','marker','o')
```

Note that you can give numeric arguments to the command as additional `femesh` arguments. Thus the command above could also have been written

```
fecom('TextNode',femesh('FindNode z== & x>=',0,0)))
```

See also the gartfe demo.

Info [,FEeli, Nodei]

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. InfoNodei 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')
```

Join [,el0] [group i, EName]

Join the groups *i* or all the groups of type *EName*. JoinAll joins all the groups that have the same element name. By default this operation is applied to FEelt but you can apply it to FEel0 by adding the el0 option 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).

```
femesh('Reset'); femesh(';Test2bay;PlotElt');
% Join using group ID
femesh('InfoElt');   % 2 groups at this step
femesh JoinGroup1:2  % 1 group now
% Join using element name
femesh('Reset'); femesh('Test2bay;PlotElt');
femesh Joinbeam1     % 1 group now
```

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.

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.

```
femesh('Reset')
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];
prop=[100 100 1.1 0 0]; % MatId ProId nx ny nz
femesh('ObjectBeamLine',1:4,prop);femesh('AddSel');
%or femesh(';ObjectBeamLine 1 2 0 2 3 0 3 4;AddSel');
% or femesh('ObjectBeamLine',1:4);
femesh('ObjectMass',3,[1.1 1.1 1.1])
femesh AddSel
femesh PlotElt; 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*'. See feutil ObjectHoleInPlate for more details.

```
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')
```

ObjectHoleInBlock

Create a hexa8 *mesh of a hole in a rectangular block.* The format is 'ObjectHoleInBlock *x0 y0 z0 nx1 ny1 nz1 nx3 ny3 nz3 dim1 dim2 dim3 r nd1 nd2 nd3 ndr*'. See feutil ObjectHoleInBlock for more details.

```
femesh('Reset')
femesh('ObjectHoleInBlock 0 0 0  1 0 0  0 1 1  2 3 3 .7  8 8 3 2')
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 `femesh('ObjectQuad 10 11',nodes,4,2)` returns model with 4 and 2 divisions in each direction with a `MatId` 10 and a `ProId` 11.

```
femesh('reset');
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
femesh('Objectquad 1 1',node,4,3); % creates model
femesh('AddSel');femesh('PlotElt')

node = [3  0  0; 5  0  0; 5  2  0; 3  2  0];
femesh('Objectquad 2 3',node,3,2); % matid=2, proid=3
femesh('AddSel');femesh('PlotElt');femesh Info
```

Divisions may be specified using a vector between `[0,1]` :

```
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
femesh('Objectquad 1 1',node,[0 .2 .6 1],linspace(0,1,10));
femesh('PlotEl0');
```

Other supported object topologies are beams and hexahedrons. For example

```
femesh('Reset')
node = [0  0  0; 2  0  0;1  3  0; 1  3  1];
femesh('Objectbeam 3 10',node(1:2,:),4); % creates model
femesh('AddSel');
femesh('Objecthexa 4 11',node,3,2,5); % creates model
femesh('AddSel');
femesh PlotElt; femesh Info
```

`Object [Arc, Annulus, Circle,Cylinder,Disk]`

Build selected object in `FEel0`. See `feutil Object` for a list of available objects. For example:

```
femesh('Reset')
femesh(';ObjectArc 0 0 0 1 0 0 0 1 0 30 1;AddSel');
femesh(';ObjectArc 0 0 0 1 0 0 0 1 0 30 1;AddSel');
femesh(';ObjectCircle 1 1 1 2 0 0 1 30;AddSel');
femesh(';ObjectCircle 1 1 3 2 0 0 1 30;AddSel');
femesh(';ObjectCylinder 0 0 0  0 0 4 2 10 20;AddSel');
```

```
femesh(';ObjectDisk 0 0 0 3 0 0 1 10 3;AddSel');
femesh(';ObjectAnnulus 0 0 0 2 3 0 0 1 10 3;AddSel');
femesh('PlotElt')
```

Optim [Model, NodeNum, EltCheck]

OptimModel removes nodes unused in FEelt *from* FEnode.

OptimNodeNum does a permutation of nodes in FEnode such that the expected matrix bandwidth is smaller. This is only useful to export models, since here DOF renumbering is performed by fe_mk.

OptimEltCheck attempts to fix geometry pathologies (warped elements) in quad4, hexa8 and penta6 elements.

Orient, Orient *i* [, n *nx ny nz*]

Orient elements. For volumes and 2-D elements which have a defined orientation, femesh('Orient') calls element functions with standard material properties to determine negative volume orientation and permute nodes if needed. This is in particular needed when generating models via Extrude or Divide operations which do not necessarily result in appropriate orientation (see integrules). When elements are too distorted, you may have a locally negative volume. A warning about warped volumes is then passed. You should then correct your mesh. Note that for 2D meshes you need to use 2D topology holders q4p, t3p, ...

Orient normal of shell elements. For plate/shell elements (elements with parents of type quad4, quadb or tria3) in groups *i* of FEelt, this command computes the local normal and checks whether it is directed towards the node located at *nx ny nz*. If not, the element nodes are permuted so that a proper orientation is achieved. A -neg option can be added at the end of the command to force orientation away rather than towards the nearest node.

femesh('Orient *i*',node) can also be used to specify a list of orientation nodes. For each element, the closest node in node is then used for the orientation. node can be a standard 7 column node matrix or just have 3 columns with global positions.

For example

```
% Init example
femesh('Reset'); femesh(';Testquad4;Divide 2 3;')
FEelt=FEel0; femesh('DivideGroup1 withnode1');
% Orient elements in group 2 away from [0 0 -1]
femesh('Orient 2 n 0 0 -1 -neg');
```

`Plot [Elt, El0]`

Plot selected model. `PlotElt` calls `feplot` to initialize a plot of the model contained in `FEelt`. `PlotEl0` does the same for `FEel0`. This command is really just the declaration of a new model using `feplot('InitModel',femesh('Model'))`.

Once the plot initialized you can modify it using `feplot` and `fecom`.

`Lin2quad, Quad2Lin, Quad2Tria, etc.`

Basic element type transformations.

Element type transformation are applied to elements in `FEel0`. See `feutil Lin2Quad` fore more details and a list of transformations.

```
% create 4 quad4
femesh(';Testquad4;Divide 2 3');
femesh('Quad2Tria'); % conversion
femesh PlotEl0
% create a quad, transform to triangles, divide each triangle in 4
femesh(';Testquad4;Quad2Tria;Divide2;PlotEl0;Info');
% lin2quad example:
femesh('Reset'); femesh('Testhexa8');
femesh('Lin2Quad epsl .01');
femesh('Info')
```

`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. A sample call would be

```
% create 4 quad4
femesh('Reset'); femesh(';Testquad4;Divide 2 3');
femesh('RemoveEl0 WithNode 1')
femesh PlotEl0
```

`RepeatSel` *nITE tx ty tz*

Element group translation/duplication. `RepeatSel` repeats the selected elements (`FEl0`) *nITE* times with global axis translations *tx ty tz* between each repetition of the group. If needed, new nodes are added to `FEnode`. An example is treated in the `d_truss` demo.

```
femesh('Reset'); femesh(';Testquad4;Divide 2 3');
femesh(';RepeatSel 3 2 0 0'); % 3 repetitions, translation x=2
femesh PlotEl0
% alternate call:
%                                          number, direction
% femesh(sprintf(';repeatsel %f %f %f %f', 3,    [2 0 0]))
```

`Rev` *nDiv OrigID Ang nx ny nz*

Revolution of selected elements in `FEl0`. See `feutil Rev` for more details. For example:

```
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
%     divi origin angle direct
%r1 = [40  0 0 0  360   0 1 0];
%femesh(sprintf('Rev %f o %f %f %f %f %f %f',r1))
```

`RotateSel` *OrigID Ang nx ny nz*

Rotation. The selected elements `FEl0` 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 `FEl0`

```
femesh('Reset');
```

```
femesh(';Testquad4;Divide 2 3');
% center is node 1, angle 30, aound axis z
%                                        Center angle dir
st=sprintf(';RotateSel %f %f %f %f %f',[1      30   0 0 1]);
femesh(st);  femesh PlotEl0
fecom(';Triax;TextNode'); axis on
```

Sel [Elt,El0] *ElementSelectors*

Element selection. SelElt places in the selected model FEel0 elements of FEelt that verify certain conditions. You can also select elements within FEel0 with the SelEl0 command. Available element selection commands are described under the FindElt command and section 7.12 .

femesh('SelElt *ElementSelectors*').

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('SetGroup1:3 pro 4')
femesh('SetGroup 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.

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.

Test

Some unique element model examples. See list with femesh('TestList'). For example a simple cube model can be created using

```
model=femesh('TestHexa8'); % hexa8 test element
```

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('Reset');
femesh(';Testquad4;Divide 2 3;AddSel');
femesh(';TransSel 3 1 0;AddSel'); % Translation of [3 1 0]
femesh PlotElt
fecom(';Triax;TextNode')
```

UnJoin *Gp1 Gp2*

Duplicate nodes which 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('Reset'); femesh('Test2bay');
femesh('FindNode group1 & group2') % nodes 3 4 are common
femesh('UnJoin 1 2');
femesh('FindNode group1 & group2') % no longer any common node
```

A more general call allows to separate nodes that are common to two sets of elements `femesh('UnJoin','Selection1','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 4.5 , demos `gartfe`, `d_ubeam`, `beambar` ...

feutil

Purpose

Finite element mesh handling utilities.

Syntax

```
[out,out1] = feutil('CommandString',model,...)
```

Description

feutil provides a number of tools for mesh creation and manipulation.

Some commands return the model structure whereas some others return only the element matrix. To mesh a complex structure one can mesh each subpart in a different model structure (model, mo1, ...) and combine each part using AddTest command. To handle complex model combination (not only meshes but whole models with materials, bases, ...), one can use the CombineModel command.

Available feutil commands are

Advanced

Advanced command with non trivial input/output formats or detailed options are listed under feutila.

AddElt

```
model.Elt=feutil('AddElt',model.Elt,'EltName',data)
```

This command can be used to add new elements to a model. EltName gives the element name used to fill the header. data describes elements to add (one row per element). Following example adds celas elements to the basis of a simple cube model.

```
% Adding elements to a model
femesh('Reset'); model=femesh('Testhexa8'); % simple cube model
data=[1 0 123 0 0 1 1e3; 2 0 123 0 0 1 1e3;
      3 0 123 0 0 1 1e3; 4 0 123 0 0 1 1e3]; % n1 n2 dof1 dof2 EltId ProId k
model.Elt=feutil('AddElt',model.Elt,'celas',data);
cf=feplot(model);
```

AddNode[,New] [, From i] [,epsl *val*]

`[AllNode,ind]=feutil('AddNode',OldNode,NewNode);`

Combine (without command option `New`) or *append* (with command option `New`) `NewNode` to `OldNode`. Without command option `New`, `AddNode` combines `NewNode` to `OldNode`: it finds nodes in `NewNode` that coincide with nodes in `OldNode` and appends other nodes to form `AllNode`. With command option `New`, `AddNode` simply appends `NewNode` to `OldNode`.

`AllNode` is the new node matrix with added nodes. `ind` (optional) gives the indices of the `NewNode` nodes in the `AllNode` matrix.

`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`.

SDT uses an optimized algorithm available in `feutilb`.

By default, nodes that repeated in `NewNode` are coalesced onto the same node (a single new node is added). If there is not need for that coalescence, you can get faster results with `AddNode-nocoal`.

`ind=feutilb('AddNode -near epsl value',n1,n2);` returns a sparse matrix with non zero values in a given colum indicating of `n1` nodes that are within `epsl` of each `n2` node (rows/columns correspond to `n1/n2` node numbers).

`id=feutilb('AddNode -nearest epsl value',n1,xyz);` returns vector giving the nearest `n1 NodeId` to each `xyz` node the search area being limited to `epsl`. When specified with a 7 column `n2`, the result is `sparse(n2(:,1),1,n1_index)`. For fine meshes the algorithm can use a lot of memory. If `n2` is not too large it is then preferable to use an `AddNode` command with a tolerance sufficient for a match `[n3,ind]=feutil('AddNode epsl value',n1,n2);id=n3(ind,1)`.

AddSet[NodeId, EltId, FaceId, EdgeId]

Command `AddSet` packages the generation of sets in an SDT model. Depending on the type of set several command options can apply.

- `model=feutil('AddSetNodeId',model,'name','FindNodeString')` adds the selection `FindNodeString` as a `set` of nodes `name` to `model`. `FindNodeString` can be replaced by a column vector of `NodeId`.
- Syntax is the same for `AddSetEltId` with a `FindEltString` selection. `FindEltString` can be replaced by a column vector of `EltId`. Command option `FromInd` allows providing element indices instead of IDs.

- For faces with AddSetFaceId, the element selection argument FindEltString must result in the generation of a face selection. One can use the SelFace token in the FindEltString to this purpose. As an alternative, one can directly provide an element matrix resulting from a SelFace selection, or a 2 column list of respectively EltId and Face identifiers. For face identifier conversion to other code conventions, one can use command option @*fun* to obtain a set with a ConvFcn set to *fun*, see set for more details.

- For generation of EdgeId sets, the element selection argument FindEltString must result in the generation of an edge selection. One can use the SelEdge token in the FindEltString to this purpose. As an alternative, one can directly provide an element matrix resulting from a SelEdge selection, or a 2 column list of respectively EltId and Edge identifiers. Support for edge identifier conversion and setname selection is not provided yet.

The option -id *value* can be added to the command to specify a set ID.

By default the generated set erases any previously existing set with the same name, regardless of the type. Command option New alters this behavior by incrementing the set name. One can use the command second output to recover the new name.

Command option -Append allows generation of a meta-set. The meta-set is an agglomeration of several sets of possibly various types, see set for more information.

- The base syntax requires providing the meta-set name and the set name. model=feutil('AddSetEltId -Append',model,'name','FindEltString','subname') will thus add the elements found as a sub set named subname of meta-set name. subname can be a 1x2 cell array {subname,subgroup} providing the set name and the set subgroup it belongs to. By default subgroup is set to the set type.

- Generation of a meta-set gathering all base sets in the model is possible by omitting subname and the FindEltString.

By default command AddSet returns the model as a first output and possibly the set data structure in a second output. Command option -get alters this behavior returning the data set structure without adding it to the model. For FaceId or EdgeId sets, command option -get can output the elements selected by the FindEltString.

Following example defines a set of each type on the ubeam model:

```
% Defining node elements or face sets in a model
model=demosdt('demo ubeam');
% Add a set of NodeId, and recover set data structure
```

```
[model,data]=feutil('AddSetNodeId',model,'nodeset','z==1');
% Add a set of EltId
model=feutil('AddSetEltId -id18',model,'eltset','WithNode{z==0}');
% Generate a set of EltId without model addition
data=feutil('AddSetEltId -id18 -get',model,'eltset','WithNode{z==0}');
% Generate a set of FaceId
model=feutil('AddSetFaceId',model,'faceset','SelFace & InNode{z==0}');
% Generate a set of FaceId without model addition
[data1,elt]=feutil('AddSetFaceId -get',model,'faceset','SelFace & WithNode{z==0}');

% Sample visalization commands
cf=feplot; % get feplot handle
[elt,ind]=feutil('FindElt setname  eltset',model); % FindElt based on set name
cf.sel='setname faceset'; % element selection based on a FaceId set

% Lower level set handling
% Generate a FaceSet from an EltSet
r1=cf.Stack{'eltset'};r1.type='FaceId';r1.data(:,2)=1;
cf.Stack{'set','faceset'}=r1;
% Generate a DOF set from a node set
r1=cf.Stack{'nodeset'};r1.type='DOF';r1.data=r1.data+0.02;
cf.Stack{'set','dofset'}=r1;
% Visualize set data in promodel stack
fecom(cf,'curtab Stack','eltset');
```

AddTest[,-EGID *i*][,*NodeShift*,Merge,Combine]

`model=feutil('AddTest',mo1,mo2);` *Combine 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 `model`.

Note that the earlier objective of combining test and FEM models is now more appropriately dealt with using `SensDof` entries, see section 4.6 for sensor definitions and section 3.1 for test/analysis correlation. If you aim at combining several finite element models into an assembly, with proper handling of materials, element IDs, bases,..., you should rather use the more appropriate `CombineModel` command.

- **By default** the node sets are considered to be disjoint. New nodes are added starting from `max(mo1.Node(:,1))+1` or from *NodeShift*+1 if the argument is specified.

Thus `feutil('AddTest NodeShift',mo1,mo2)` adds `mo2` nodes to `mo1.Node` while adding *NodeShift* to their initial identification number. The same *NodeShift* is added to node numbers in `mo2.Elt` which is appended to `mo1.Elt`. `mo2` can be a wire frame matrix read with `ufread` for example.

- With command option `Merge` it is assumed that some nodes are common but their numbering is not coherent. Non coincident nodes (as defined by the `AddNode` command) are added to `mo1.Node` and `mo2.Elt` is renumbered according to resulting `model.Node`. Command option `Merge-Edge` is used to force mid-side nodes to be common if the end nodes are. Note that command `Merge` will also merge all coincident nodes of `mo2`.
- With command option `Combine` it is assumed that some nodes are common and their numbering is coherent. Nodes of `mo2.Node` with new `NodeId` values are added to `mo1.Node` while common `NodeId` values are assumed to be located at the same positions.
- You can specify an `EGID` value for the elements that are added using `AddTest -EGID -1` for example. In particular negative `EGID` values are display groups so that they will be ignored in model assembly operations. Command option `keeptest` allows to retain existing test frames when adding a new one. If the same `EGID` is declared, test frames are then combined in the same group.
- Command option `-NoOri` returns model without the `Info,OrigNumbering` entry in the model stack.

`Divide` *div1 div2 div3*

```
model=feutil('Divide div1 div2 div3',model);
```

Mesh refinement by division of elements. `Divide` applies to all groups in `model.Elt`. To apply the division to a selection within the model use `ObjectDivide`.

Division directions ***div1 div2 div3*** are here understood in the local element basis, thus depending on the declared node orders in the connectivity matrix that refer to the reference cell. Uneven divisions as function of the direction will thus require some care regarding the element declaration if the original mesh has been heterogeneously generated.

Currently supported divisions are

- segments : elements with `beam1` parents are divided in *div1* segments of equal length.
- quadrilaterals: elements with `quad4` or `quadb` parents are divided in a regular mesh of *div1* by *div2* quadrilaterals.
- hexahedrons: elements with `hexa8` or `hexa20` parents are divided in a regular grid of *div1* by *div2* by *div3* hexahedrons.

- tria3 can be divided with an equal division of each segment specified by *div1*.
- mass1 and celas elements are kept unchanged.

The Divide command applies element transformation schemes on the element parent topological structure. By default, the original element names are maintained. In case of trouble, element names can be controlled by declaring the proper parent name or use the SetGroupName command before and after divide.

The division preserves properties other than the node numbers, in addition final node numbering/ordering will depend on the MATLAB version. It is thus strongly recommended not to base meshing scripts on raw NodeId.

You can obtain unequal divisions by declaring additional arguments whose lines give the relative positions of dividers. Note that this functionality has not been implemented for quadb and tria3 elements.

For example, an unequal 2 by 3 division of a quad4 element would be obtained using `model=feutil('divide',[0 .1 1],[0 .5 .75 1],model)` (see also the gartfe demo).

```
% Refining a mesh by dividing the elements
% Example 1 : beam1
femesh('Reset'); model=femesh('Testbeam1');  % build simple beam model
model=feutil('Divide 3',model); % divide by 3
cf=feplot(model); fecom('TextNode'); % plot model and display NodeId

% Example 2 : you may create a command string
femesh('Reset'); model=femesh('Testbeam1');  % build simple beam model
number=3;
st=sprintf('Divide %f',number);
model=feutil(st,model);
cf=feplot(model); fecom('TextNode')

% Example 3 : you may use uneven division
femesh('Reset'); model=femesh('Testquad4');  % one quad4 created
model=feutil('Divide',model,[0 .1 .2 1],[0 .3 1]);
feplot(model);
```

An inconsistency in division for quad elements was fixed with version 1.105, you can obtain the consistent behavior (first division along element x) by adding the option -new anywhere in the divide command.

DivideInGroups

`elt=feutil('DivideInGroups',model);`
Finds groups that are not connected (no common node) and places each of these groups in a single element group.

DivideGroup *i ElementSelectors*

`elt=feutil('DivideGroup i ElementSelector',model);`

Divides a single group *i* in two element groups. The first new element group is defined based on the element selectors (see section 7.12).

For example `elt=feutil('divide group 1 withnode{x>10}',model);`

EltId

`[EltId]=feutil('EltId',elt)` returns the element identifier for each element in `elt`. It currently does not fill `EltId` for elements which do not support it.
`[EltId,elt]=feutil('EltIdFix',elt)` returns an `elt` where the element identifiers have been made unique.
Command option `-elt` can be used to set new `EltId`.
Command option `-model` can be used to set new `EltId` and renumber model Stack data, a model structure must be input, and the output is then the model.

```
% Handling elements IDs, renumbering elements
model=femesh('TestHexa8')
[EltId,model.Elt]=feutil('EltIdFix',model.Elt); % Fix and get EltId
[model.Elt,EltIdPos]=feutil('eltid-elt',model,EltId*18); % Set new EltId
model.Elt(EltIdPos>0,EltIdPos(EltIdPos>0)) % New EltId

% Renumber EltId with stack data
model=feutil('AddSetEltId',model,'all','groupall');
model=feutil('EltId-Model',model,EltId+1);
```

Extrude *nRep tx ty tz*

Extrusion. Nodes, lines or surfaces of model 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 extrusion. For example,
`model=feutil('Extrude 0 0 0 1',model,[0 logspace(-1,1,5)])` will create an exponentially spaced mesh in the z direction. The second argument gives the positions of the sections for an axis such that `tx ty tz` is the unit vector.

```
% Extruding mesh parts to build a model
% Example 1 : beam
femesh('Reset'); model=femesh('Testbeam1'); % one beam1 created
model=feutil('Extrude 2 1 0 0',model); % 2 extrusions in x direction
cf=feplot(model);

% Example 2 : you may create the command string
number=2;step=[1 0 0];
st=sprintf('Extrude %f %f %f %f',[number step]);
femesh('Reset'); model=femesh('Testbeam1'); % one beam1 created
model=feutil(st,model);
cf=feplot(model);

% Example 3 : you may uneven extrusions in z direction
femesh('Reset'); model=femesh('Testquad4');
model=feutil('Extrude 0 0 0 1',model,[0 .1 .2 .5 1]);
     % 0 0 0 1        :  1 extrusion in z direction
     % [0 .1 .2 .5 1] :  where extrusions are made
feplot(model)
```

GetDof *ElementSelectors*

Command to obtain DOF from a model, or from a list of `NodeId` and DOF.

Use `mdof=feutil('GetDof',dof,NodeId);` to generate a DOF vector from a list of DOF indices `dof`, a column vector (*e.g.* `dof=[.01;.02;.03]`), and a list of `NodeId`, a column vector. The result will be sorted by DOF, equivalent to `mdof = [NodeId+dof(1);NodeId+dof(2);...]`.

Call `mdof=feutil('GetDof',NodeId,dof);` will output a DOF vector sorted by `NodeId`, equivalent to `mdof = [NodeId(1)+dof;NodeId(2)+dof;...]`.

The nominal call to get DOFs used by a model is `mdof=feutil('GetDOF',model)`. These calls are performed during assembly phases (`fe_mk`, `fe_load`, ...). This supports elements with variable DOF numbers defined through the element rows or the element property rows. To find DOFs of a part of the model, you should add a `ElementSelector` string to the `GetDof` command string.

Note that node numbers set to zero are ignored by `feutil` to allow elements with variable number of nodes.

`FindElt` *ElementSelectors*

Find elements based on a number of selectors described in section 7.12 . The calling format is

```
[ind,elt] = feutil('FindElt ElementSelector',model);
```

where `ind` gives the row numbers of the elements in `model.Elt` (but not the header rows except for unique superelements which are only associated to a header row) and `elt` (optional) 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. See also `SelElt`, `RemoveElt` and `DivideGroup`, the `gartfe` demo, `fecom` selections.

`FindNode` *Selectors*

Find node numbers based on a number of node 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).

The calling format is
`[NodeId,Node] = feutil('FindNode NodeSelector',model);`

Output arguments are the `NodeId` of the selected nodes and the selected nodes `Node` as a second optional output argument.

As an example you can show node numbers on the right half of the `z==0` plane using the commands

```
fecom('TextNode',feutil('FindNode z==0 & x>0',model))
```

Following example puts markers on selected nodes

```
% Finding nodes and marking/displaying them in feplot
demosdt('demo ubeam'); cf=feplot; % load U-Beam model
fecom('ShowNodeMark',feutil('FindNode z>1.25',cf.mdl),'color','r')
fecom('ShowNodeMark-noclear',feutil('FindNode x>0.2*z|x<-0.2*z',cf.mdl),...
      'color','g','marker','o')
```

Note that you can give numeric arguments to the command as additional `feutil` arguments. Thus the command above could also have been written `feutil('FindNode z== & x>=',0,0))`

See also the gartfe demo.

GetEdge[Line,Patch]

These feutil commands are used to create a model containing the 1D edges or 2D faces of a model. A typical call is

```
% Generate a contour (nD-1) model from a nD model
femesh('reset'); model=femesh('Testubeam');
elt=feutil('GetEdgeLine',model); feutil('infoelt',elt)
```

GetEdgeLine supports the following variants MatId retains inter material edges, ProId retains inter property edges, Group retains inter group edges, all does not eliminate internal edges, InNode only retains edges whose node numbers are in a list given as an additional feutil argument.

These commands are used for SelEdge and SelFace element selection commands. Selface preserves the EltId and adds the FaceId after it to allow face set recovery.

GetElemF

Header row parsing. In an element description matrix, element groups are separated by header rows (see section 7.2) which for the current group jGroup is given by elt(EGroup(jGroup),:) (one can obtain EGroup - the positions of the headers in the element matrix - using [EGroup,nGroup]=getegroup(model.Elt)). 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.

Get[Line,Patch]

Line=feutil('GetLine',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.

Patch=feutil('GetPatch',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 ni 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.

The `all` option skips the internal edge/face elimination step. These commands are used in wire-frame and surface rendering.

GetNode *Selectors*

`Node=feutil('GetNode Selectors',model)` returns a matrix containing nodes rather than NodeIds obtained with the `FindNode` command. The indices of the nodes in `model.Node` can be returned as a 2nd optional output argument. This command is equivalent to the `feutil` call

`[NodeId,Node]=feutil('FindNode Selectors',model)`.

GetNormal[Elt,Node][,Map],GetCG

`[normal,cg]=feutil('GetNormal[elt,node]',model)` returns normals to elements/nodes in `model`. `CG=feutil('GetCG',model)` returns the CG locations. Command option `-dir i` can be used to specify a local orientation direction other than the normal (this is typically used for composites). `MAP=feutil('getNormal Map',model)` returns a data structure with the following fields

`ID`	column of identifier (as many as rows in the `.normal` field). For `.opt=2` contains the `NodeId`. For `.opt=1` contains the `EltId`.
`normal`	$N \times 3$ where each row specifies a vector at `ID` or `vertex`.
`opt`	1 for MAP at element center, 2 for map at nodes.
`color`	$N \times 1$ optional real value used for color selection associated with the axes color limits.
`DefLen`	optional scalar giving arrow length in plot units.

The `MAP` data structure may be viewed using

`fecom('ShowMap',MAP);fecom('ScaleOne');`

Info[,Elt, Node*i*]

`feutil('Info',model);` *Information on model.* `Info` by itself gives general information about `model`. `InfoNode`*i* gives information about all elements that are connected to node of NodeId *i*.

Join[group *i*, *EltName*]

Join the groups *i* or all the groups of type *EltName*. JoinAll joins all the groups that have the same element name. Note that with the selection by group number, you can only join groups of the same type (with the same element name). JoinAll joins all groups with identical element names.

You may join groups using there ID

```
% Joining groups of similar element types
femesh('Reset'); model=femesh('Test2bay');
% Join using group ID
feutil('Info',model);   % 2 groups at this step
model=feutil('JoinGroup1:2',model)  % 1 group now
feutil('Info',model);
% Join using element types
% Note you can give model (above) or element matrix (below)
femesh('Reset'); model=femesh('Test2bay');
model.Elt=feutil('Joinbeam1',model.Elt);  % 1 group now
```

Matid, ProId, MPID

MatId=feutil('MatId',model) returns the element material identifier for each element in model.Elt. One can also modify MatId of the model giving a third argument. model=feutil('MatId',model,r1) r1 can be a global shift on all non zero MatId or a matrix whose first column gives old MatId and second new MatId. MatId renumbering is applyed to elements, model.pl and model.Stack 'mat' entries. The ProId command works similarly.

MPId returns a matrix with three columns MatId, ProId and group numbers.
model.Elt=feutil('mpid',model,mpid) can be used to set properties of elements in model.Elt matrix.

Node[trans, rot, mir, DefShift]

The command feutil('node [trans,rot,mir]',model,RO) allows to move model nodes (or part of a model with a provided selection) with standard transformations :

- translation : trans *x y z*
- rotation : rot *x1 x2 x3 n1 n2 n3 theta* with *xi* the coordinate of the node and *ni* the direction of the axe and *theta* the angle in degree

- plane symmetry :
  - plane x y or z : `mir x`, `mir y` or `mir z`
  - point + normal : `mir "o x1 x2 x3 n1 n2 n3"` with xi the coordinate of the node and ni the direction of the normal to the plane
  - plane equation : `mir "eq a b c d"` defining the plane $aX + bY + cZ + d = 0$
  - best plane defined by list of node coordinates :
    `feutil('node mir',model,struct('node',[x1 y1 z1;x2 y2 z2;...]))`
  - best plane defined by list of nodeids : `mir "nodeid id1 id2 id3"`
- rigid body matrix : `feutil('node',model,struct('rb',[4x4 RB matrix]))`

For each call, it is possible to either provide inputs as text string or as structure given on third argument with the field name corresponding to the wanted transformation.

An element selection can be provided in the text command (`sel"EltSel"`) or as a text in a .sel field of the `RO` stucture to apply the transformation on only a part of the model. See `FindElt`.

Here is an exhaustive list of examples

```
model=femesh('test tetra4'); % Load model wontaining a tetrahedron
model.Node=feutil('addnode',model.Node,[0-1 0 0]); % Add a node
model.Elt=feutil('addelt',model.Elt,'mass1',5); % Set this node as a mass1 element
feplot(model); % Display
% Displacement transformations
% translation in the direction [1 0 0] sepcified in the text command
model=feutil('node trans 1 0 0',model); feplot(model);
% rotation of 180deg arround the axis defined by node [1 0 0] and vector [0 0 1]
RO=struct('rot',[1 0 0  0 0 1 180]); % rotation is the last number
% Only nodes in "group1" are moved
model=feutil('node -sel"group1"',model,RO); feplot(model);
% Rigid body transformation (matrix in field rb) on nodes in group1
RO=struct('rb',[1 0 0 -1;0 1 0 0;0 0 1 0;0 0 0 1],'sel','group1');
model=feutil('node',model,RO); feplot(model);
% mirror transformation
% Plane y=0
model=feutil('node mir y',model); feplot(model);
% Same plane definined with node [0 0 0] and normal [0 1 0]
model=feutil('node mir "o 0 0 0 0 1 0"',model); feplot(model);
% Same plane definined with nodeid 1 2 4
model=feutil('node mir "nodeid 1 2 4"',model); feplot(model);
```

```
% Same plane defininied with equation 0*x+1*y+0*z+0=0, given as last
% argument in a structure
RO=struct('eq',[0 1 0 0]);
model=feutil('node mir',model,RO); feplot(model);
% Mirror with respect to the "best" plane passing through the node list
RO=struct('node',[0 -0.1 0;1 0 0;0 0 1;1 0.3 1],'sel','group1');
model=feutil('node mir',model,RO); feplot(model);
fecom('shownodemark',[0 -0.1 0;1 0 0;0 0 1;1 0.3 1]); % Show nodes defining the plane
```

ObjectBeamLine *i*, ObjectMass *i*

`elt=feutil('ObjectBeamLine 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 `elt` as a single group of `beam1` elements.

For example `elt=feutil('ObjectBeamLine 1:3 0 4 5')` creates a group of three `beam1` elements between nodes `1 2`, `2 3`, and `4 5`.

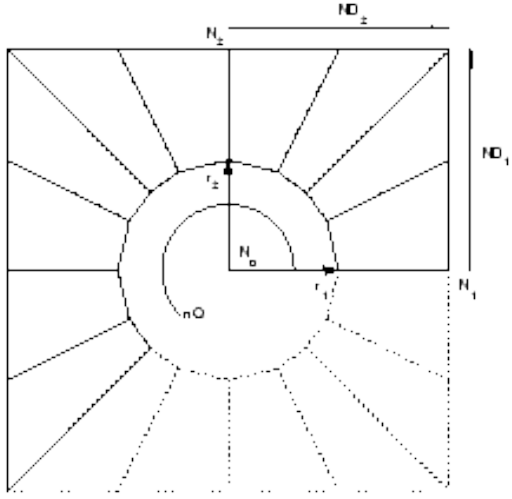
An alternate call is `elt=feutil('ObjectBeamLine',ind)` where `ind` is a vector containing the node numbers. You can also specify a element name other than `beam1` and properties to be placed in columns 3 and more using `elt=feutil('ObjectBeamLine -EltName',ind,prop)`.

`elt=feutil('ObjectMass 1:3')` creates a group of concentrated `mass1` elements at the declared nodes.

```
% Build a mesh by addition of defined beam lines and masses
model=struct('Node',[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], 'Elt',[]);
prop=[100 100 1.1 0 0]; % MatId ProId nx ny nz
model.Elt=feutil('ObjectBeamLine 1 2 0 2 3 0 3 4',prop);
% or model.Elt=feutil('ObjectBeamLine',1:4);
model.Elt=feutil('ObjectMass',model,3,[1.1 1.1 1.1]);
%model.Elt(end+1:end+size(elt,1),1:size(elt,2))=elt;
feplot(model);fecom textnode
```

ObjectHoleInPlate

```
model=feutil('ObjectHoleInPlate ...',model);
```



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).

```
% Build a model of a plate with a hole
model=struct('Node',[1 0 0 0  0 0 0; 2 0 0 0  1 0 0; 3 0 0 0  0 2 0],'Elt',[]);
model=feutil('ObjectHoleInPlate 1 2 3 .5 .5 3 4 4',model);
model=feutil('Divide 3 4',model); % 3 divisions around, 4 divisions along radii
feplot(model)
% You could also use the call
model=struct('Node',[1 0 0 0  0 0 0; 2 0 0 0  1 0 0; 3 0 0 0  0 2 0],'Elt',[]);
%   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);
model=feutil(st,model);
```

ObjectHoleInBlock

`model=feutil('ObjectHoleInBlock ...');` *Create a* hexa8 *mesh of a hole in a rectangular block.* The format is *'ObjectHoleInBlock x0 y0 z0 nx1 ny1 nz1 nx3 ny3 nz3 dim1 dim2 dim3 r nd1 nd2 nd3 ndr'* giving the center of the block (*x0 y0 z0*), the directions along the first and third dimensions of the block (*nx1 ny1 nz1 nx3 ny3 nz3*, third dimension is along the hole), the 3 dimensions (*dim1 dim2 dim3*), the radius of the cylinder hole (*r*), the number of divisions of each dimension of the cube (*nd1 nd2 nd3*, the 2 first should be even) and the number of divisions along the radius (*ndr*).

```
% Build a model of a cube with a cylindrical hole
model=feutil('ObjectHoleInBlock 0 0 0  1 0 0  0 1 1  2 3 3 .7  8 8 3 2')
```

Object[Quad,Beam,Hexa] *MatId ProId*

`model=feutil('ObjectQuad MatId ProId',model,nodes,div1,div2)` *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 (`div1` and `div2`). The result is a regular mesh.

For example `model=feutil('ObjectQuad 10 11',nodes,4,2)` returns model with 4 and 2 divisions in each direction with a `MatId` 10 and a `ProId` 11.

An alternate call is `model=feutil('ObjectQuad 1 1',model,nodes,4,2)`: the quadrangular mesh is added to the model.

```
% Build a mesh based on the refinement of a single quad element
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
model=feutil('Objectquad 1 1',node,4,3); % creates model

node = [3  0  0; 5  0  0; 5  2  0; 3  2  0];
model=feutil('Objectquad 2 3',model,node,3,2); % matid=2, proid=3
feplot(model);
```

Divisions may be specified using a vector between `[0,1]` :

```
% Build a mesh based on the custom refinement of a single quad element
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 hexahedrons. For example

```
% Build a mesh based on the custom refinement of a single element
node = [0  0  0; 2  0  0;1  3  0; 1  3  1];
model=feutil('Objectbeam 3 10',node(1:2,:),4); % creates model
model=feutil('Objecthexa 4 11',model,node,3,2,5); % creates model
feutil('infoelt',model)
```

Object [Arc, Annulus, Circle, Cylinder, Disk]

These object constructors follow the format

`model=feutil('ObjectAnnulus x y z r1 r2 nx ny nz Nseg NsegR',model)` with `x y z` the coordinates of the center, `nx ny nz` the coordinates of the normal to the plane containing the annulus, `Nseg` the number of angular subdivisions, and `NsegR` the number of segments along the radius. The resulting model is in `quad4` elements.

`model=feutil('ObjectArc x y z x1 y1 z1 x2 y2 z2 Nseg obt',model)` with `x y z` the coordinates of the center, `xi yi zi` the coordinates of the first and second points defining the arc boundaries, `Nseg` the number of angular subdivisions, and `obt` for obtuse, set to `1` to get the shortest arc between the two points or `-1` to get the complementary arc. The resulting model is in `beam1` elements.

`model=feutil('ObjectCircle xc yc zc r nx ny nz Nseg',model)` with `xc yc zc` the coordinates of the center, `r` the radius, `nx ny nz` the coordinates of the normal to the plane containing the circle, and `Nseg` the number of angular subdivisions. The resulting model is in `beam1` elements.

`model=feutil('ObjectCylinder x1 y1 z1 x2 y2 z2 r divT divZ',model)` with `xi yi zi` the coordinates of the centers of the cylinder base and top circles, `r` the cylinder radius, `divT` the number of angular subdivisions, and `divZ` the number of subdivisions in the cylinder height. The resulting model is in `quad4` elements.

`model=feutil('ObjectDisk x y z r nx ny nz Nseg NsegR',model)` with `x y z`, the coordinates of the center, `r` the disk radius, `nx ny nz` the coordinates of the normal to the plane containing the disk, `Nseg` the number of angular subdivisions, and `NsegR` the number of segments along the radius. The resulting model is in `quad4` elements. Command option `-nodeg` avoids degenerate elements by transforming them into `tria3` elements.

For example:

```
% Build a mesh based on simple circular topologies
model=feutil('object arc 0 0 0 1 0 0 0 1 0 30 1');
model=feutil('object arc 0 0 0 1 0 0 0 1 0 30 1',model);
model=feutil('object circle 1 1 1 2 0 0 1 30',model);
model=feutil('object circle 1 1 3 2 0 0 1 30',model);
model=feutil('object cylinder 0 0 0  0 0 4 2 10 20',model);
model=feutil('object disk 0 0 0 3 0 0 1 10 3',model);
model=feutil('object disk -nodeg 1 0 0 3 0 0 1 10 3',model);
model=feutil('object annulus 0 0 0 2 3 0 0 1 10 3',model);
feplot(model)
```

ObjectDivide

Applies a `Divide` command to a selection within the model. This is a packaged call to `RefineCell`, one thus has access to the following command options:

- `-MPC` to generate `MPC` constraints to enforce displacement continuity at non conforming interfaces
- `KnownNew` to add new nodes without check

- -noSData asks no to add model stack entry info,newcEGI that provides the indices of new elements in model.

```
% Perform local mesh refinement
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
model=feutil('Objectquad 1 1',node,4,3); % creates model
model=feutil('ObjectDivide 3 2',model,'WithNode 1');
feplot(model);

% Perform a non uniform local mesh refinement with MPC
node = [0  0  0; 2  0  0; 2  3  0; 0  3  0];
model=feutil('Objectquad 1 1',node,4,3); % creates model
model=feutil('ObjectDivide 3 2 -MPC',model,...
 'WithNode 1',[0 .2 1],[0 .25 .8 1]);
% display model and MPC constraint
feplot(model);
fecom(';promodelinit;proviewon;')
fecom('curtabCases','MPCedge');
```

Optim[Model, NodeNum, EltCheck]

model.Node=feutil('Optim...',model);
model.Node=feutil('OptimModel',model) removes nodes unused in model.Elt from model.Node. This command is very partial, a thorough model optimization is obtained using feutilb SubModel with groupall selection. model=feutilb('SubModel',model,'groupall');. To recover used nodes the most complete command is feutilb GetUsedNodes.

model.Node=feutil('OptimNodeNum',model) does a permutation of nodes in model.Node such that the expected matrix bandwidth is smaller. This is only useful to export models, since here DOF renumbering is performed by fe_mk.
model=feutil('OptimEltCheck',model) attempts to fix geometry pathologies (warped elements) in quad4, hexa8 and penta6 elements.

model=feutil('OptimDegen',model) detects degenerate elements and replaces them by the proper lower node number case hexa -> penta.

Orient, Orient *i* [, n *nx ny nz*]

Orient elements. For volumes and 2-D elements which have a defined orientation
model.Elt=feutil('Orient',model) calls element functions with standard material properties to

determine negative volume orientation and permute nodes if needed. This is in particular needed when generating models via Extrude or Divide operations which do not necessarily result in appropriate orientation (see integrules). When elements are too distorted, you may have a locally negative volume. A warning about warped volumes is then passed. You should then correct your mesh.

Note that for 2D meshes you need to use 2D element names (q4p, t3p, ...) rather than quad4, tria3, ... Typically model.Elt=feutil('setgroup1 name q4p',model).

Orient normal of shell elements. For plate/shell elements (elements with parents of type quad4, quadb or tria3) in groups *i* of model.Elt, model.Elt=feutil('Orient *i* n *nx ny nz*',model) command computes the local normal and checks whether it is directed towards the node located at *nx ny nz*. If not, the element nodes are permuted to that a proper orientation is achieved. A -neg option can be added at the end of the command to force orientation away rather than towards the nearest node.

model.Elt=feutil('Orient *i*',model,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

```
% Specify element orientation
% Load example
femesh('Reset'); model=femesh('Testquad4');
model=feutil('Divide 2 3',model);
model.Elt=feutil('Dividegroup1 WithNode1',model);
% Orient elements in group 2 away from [0 0 -1]
model.Elt=feutil('Orient 2 n 0 0 -1 -neg',model);
MAP=feutil('GetNormal MAP',model);MAP.normal
```

Quad2Lin, Lin2Quad, Quad2Tria, etc.

Basic element type transformations.

model=feutil('Lin2Quad epsl .01',model) is the generic command to generate second order meshes.
Lin2QuadCyl places the mid-nodes on cylindrical arcs.
Lin2QuadKnownNew can be used to get much faster results if it is known that none of the new mid-edge nodes is coincident with an existing node. Quad2Lin performs the inverse operation.
For this specific command many nodes become unecessary, command option -optim performs a cleanup by removing these nodes from the model, and its Stack and Case entries. Quad2Tria

searches elements for quad4 element groups and replaces them with equivalent tria3 element groups.
Hexa2Tetra replaces each hexa8 element by four tetra4 elements (this is really not a smart thing to do).
Hexa2Penta replaces each hexa8 element by six tetra4 elements (warning : this transformation may lead to incompatibilities on the triangular faces).
Penta2Tetra replaces each penta6 element by 11 tetra4 elements.

Command option KnownNew can be used for Hexa2Tetra, Hexa2Penta, and Penta2Tetra. Since these commands add nodes to the structure, quicker results can be obtained if it is known that none of the new nodes are coincident with existing ones. In a more general manner, this command option is useful if the initial model features coincident but free surfaces (*e.g.* two solids non connected by topology, when using coupling matrices). The default behavior will add only one node for both surfaces thus coupling them, while the KnownNew alternative will add one for each.

```
% Transforming elements in a mesh, element type and order
% create 2x3 quad4
femesh('Reset'); model=femesh('Testquad4');
model=feutil('Divide 2 3',model);
model=feutil('Quad2Tria',model); % conversion
feplot(model)
% create a quad, transform to triangles, divide each triangle in 4
femesh('Reset'); model=femesh('Testquad4');
model=feutil('Quad2Tria',model);
model=feutil('Divide2',model);
cf=feplot(model); cf.model
% create a hexa8 and transform to hexa20
femesh('Reset'); model=femesh('Testhexa8');
model=feutil('Lin2Quad epsl .01',model);
feutil('InfoElt',model)
```

RefineCell, Beam *l*, ToQuad

- The RefineCell command is a generic element-wise mesh refinement command. Each element can be replaced by another mesh fitted in the initial topology. This is in particular used by RefineToQuad.

  For each element type, it is possible to define an interior mesh defined in the element reference configuration. RefineCell then applies node and element additions in an optimized way to produce a final mesh in which all elements have been transformed.

  A typical syntax is `model=feutil('RefineCell',model,R1'`, with `model` a standard SDT model and `R1` a running option structure providing in particular the cell refinement topologies.

In practice, cell refinement is defined for each element type in the reference configuration, giving additional nodes by edge, then face, then volume in increasing index. New nodes are computed using an operator performing weighted sums of initial cell coordinates. If no weights are given, arithmetic average is used.

Option structure `R1` contains fields named as element types. These fields provide structures with fields

- `edge` a cell array in the format `{[newId [oldId_Av]], [weights]}` providing the nodes to be added on the edges of the initial element. It is a 1 by 2 cell array. The first part is a matrix with as many lines as new nodes to be added, the first column `newId` providing the new `NodeId` of the reference configuration and the following ones `oldId_Av` the nodes of the initial cell used to generate the new coordinates. The second part is a weight matrix, with as many lines as new nodes and as many columns as `oldId_Av` providing the weights for each node. The `weights` matrix can be left empty in which case equal weights will be used for each nodes. It can also be set a a scalar, and in this case the scalar coefficient will be used for each weight. `newId` have to be given in increasing order. This can be left blank if no node has to be added in edges.
- `face` a cell array in the same format than for field `edge`, providing the nodes to be added on the edges of the initial element. `newId` have to be given in increasing order and greater than the `edge` new IDs. This can be left blank if no node has to be added in faces.
- `volume` a cell array in the same format than for field `edge`, providing the nodes to be added in the volume of the initial element. `newId` have to be given in increasing order and greater than the `edge` new IDs and greater than the `face` new IDs. This can be left blank if no node has to be added in the volume.
- `Elt` a cell array providing the elements defined in the reference configuration topology. This is a cell array in format `{ElemP, Elt}`, `ElemP` providing the new element types and `Elt` an element matrix with no header providing the connectivies associated to `ElemP`.
- `faces` For non symmetric transformations, it is possible to define a reference node ordering of the reference configuration that allows identifying a reference face of the reference configuration.
- `shift` For non symmetric transformations, `shift` will identify the reference face in the `faces` field to allow transformation for selected faces of elements.

A sample call to refine `quad4` elements using `RefineCell` is then

```
% refine cell sample call for iso quad refinement
model=femesh('testquad4'); % base quad element
% definition of the quad transformation
R1=struct('quad4',...
```

```
 struct('edge',{{[5 1 2;6 2 3;7 3 4;8 4 1],.5}},...
 'face',{{[9 1 2 3 4],.25}},...
 'Elt',{{'quad4',[1 5 9 8;5 2 6 9;9 6 3 7;9 7 4 8]}}));
mo1=feutil('refinecell',model,R1)
[eltid,mo1.Elt]=feutil('EltIdFix;',mo1);
% Visualization
cf=feplot(mo1); fecom('textnode')
```

It is possible to restrain refinement to an element selection. This is realized by adding field `set` to `R1` containing a list of `EltId` on which the refinement will be performed.

By default, the output model only contains the refined elements. Command option `-Replace` outputs the complete model on which selected elements have been refined. In this latter case, apparition of non conforming interfaces is possible. Command option `-MPC` allows generating MPC constraints (on DOF 1,2,3) at non-conforming interfaces to enforce displacement continuity. Generated MPC are named `MPCedge` and `MPCface` respectively concerning nodes added on edges and faces.

```
% local refine cell call with MPC generation
R1.set=[1]; % define an EltId set to refine
% call for MPC for new interface edges
mo1=feutil('refinecell-replace-mpc',mo1,R1);
% display refined model and MPC
cf=feplot(mo1);
fecom(cf,';promodelinit;proviewon;curtabCase;','MPCedge');
```

Non symmetric cell refinement requires the ability to detect the element orientation regarding the reference cell orientation. The strategy implemented is based on element face (for volume) or edge (for shells) identification, through the definition in the input structure of a field `faces` providing the face indices of the reference model and a `shift` index providing a reference face. This strategy works well if the reference face is refined in a unique way. If several faces share the same refinement a yet to setup identification strategy has to be defined.

In this case, each element to be refined must be assigned a face (or edge) selection for orientation purpose. The field `set` in input structure `R1` is then mandatory with two columns, the first one providing the selected element IDs and the second one the face (or edge) identifier corresponding to the reference configuration reference face. See `feutil AddSetFaceId`, and `FindElt` commands to generate such element selection.

The following example provides a non-symmetric cell refinement of a side of a structure allowing an increase of node one side while keeping a continuous mesh.

```
% unsymmetric refine cell call
```

```
model=femesh('testquad4'); % base model
model=feutil('refineToQuad',model); % refine into 4 quad4
% fix eltid for clean element selection
[eltid,model.Elt]=feutil('EltIdFix;',model);
% define a non symmetric cell refinement
% here refinement is based on edge 1 2 using reference faces
R1=struct('quad4',...
 struct('edge',{{[5 1 2;6 1 2],...
 [2/3 1/3;1/3 2/3]}},...
 'face',{{[7 1:4;8 1:4],...
[1/6 1/3 1/3 1/6;1/3 1/6 1/6 1/3]}},...
'Elt',{{'quad4',[1 5 8 4;5 6 7 8;6 2 3 7;8 7 3 4]}},...
'faces',quad4('edge'),'shift',1));

% define a selection of edges to refine
elt=feutil('selelt seledge & innode{x==0}',model);
% here easy recovery on elements for edge selection
% based on shell element
R1.set=elt(2:end,5:6);
% call refinement
mo1=feutil('refinecell-replace',model,R1)
cf=feplot(mo1); fecom('textnode')
```

- The RefineBeam command searches model.Elt for beam elements and divides elements so that no element is longer than *l*. For beam1 elements, transfer of pin flags properties are forwarded by keeping non null flags on the new beam elements for which a pre-existing node was flagged.

  ```
  % Specific mesh refinement for beam
  femesh('Reset'); model=femesh('Testbeam1'); % create a beam
  model=feutil('RefineBeam 0.1',model);
  ```

  One can give a model subselection (FindElt command string) as 2nd argument, to refine only a part of the model beams.

- The RefineBeamUni*val* command uniformly refines all beam1elements into *val* elements. This command packages a feutil ObjectDivide call with command options KnownNew and -noSData.

  - Command option -pin allows proper pin flag forwarding for beam1 elements. transfer of pin flags properties are forwarded by keeping non null flags on the new beam elements for which a pre-existing node was flagged. This constitutes the main interest of the command.

  - Command option -MergeNew asks to merge new nodes instead of simply adding them.

- The RefineToQuad command transforms first order triangles, quadrangles, penta, tetra, and hexa to quad and hexa only while dividing each element each in two. The result is a conform mesh, be aware however that nodes can be added to your model boundaries. Using such command on model sub-parts will thus generate non conforming interfaces between the refined and non-refined parts.

  By default, new nodes are added with an AddNode command so matched new nodes are merged. Command option KnownNew allows a direct addition of new nodes without checking.

```
% Refining mesh and transforming to quadrangle elements
model=femesh('testtetra4');model=feutil('RefineToQuad',model);
feplot(model);
```

RemoveElt *ElementSelectors*

[model.Elt,RemovedElt]=feutil('RemoveElt ***ElementSelectors***',model);

Element removal. This function searches model.Elt for elements which verify certain properties selected by ***ElementSelectors*** and removes these elements from the model description matrix. 2nd output argument RemovedElt is optional and contains removed elements. A sample call would be

```
% Removing elements in a model
% create 3x2 quad4
femesh('Reset'); model=femesh('Testquad4');model=feutil('Divide 2 3',model);
[model.Elt,RemovedElt]=feutil('RemoveElt WithNode 1',model);
feplot(model)
```

Remove [Pro, Mat] *MatId, ProId*

Mat, Pro removal This function takes in argument the ID of a material or integration property and removes the corresponding entries in the model pl/il fields and in the stack mat/pro entries.

- Command option -all removes all pl/il entries found in the model and its stack.
- Command option -unused removes all pl/il entries not used by any element.

This call supports the info, Rayleigh stack entry (see sdtweb damp), so that the data entries referring to removed IDs will also be removed. By default, the non-linear properties are treated like

normal properties. Care must thus be taken if a non-linear property that is not linked to specific elements is used. Command option `-unused` will alter this behavior and keep non-linear properties.

Sample calls are provided in the following to illustrate the use.

```
% Removing material and integration properties in a model
model=femesh('testhexa8');
model=stack_set(model,'pro','integ',p_solid('default'));
model=stack_set(model,'mat','steel',m_elastic('default steel'));
model=feutil('remove pro 110',model);
model=feutil('remove pro',model,111);
model=feutil('remove mat 100',model);
model=feutil('remove mat 100 pro 1',model);
model=feutil('remove pro -all',model); % Command option -all
model=feutil('remove mat pro -all',model);
model=femesh('testhexa8'); % Command option -unused
model=feutil('remove mat pro -unused',model);
```

Renumber

`model=feutil('Renumber',model'`*`NewNodeNumbers`*`)` can be used to change the node numbers in the model. Currently nodes, elements, DOFs and deformations , nodeset, par, cyclic and other Case entries are renumbered.

`NewNodeNumbers` is the total new NodeIds vector. *`NewNodeNumbers`* can also be a scalar and then defines a global NodeId shifting. If *`NewNodeNumbers`* has two columns, first giving old NodeIds and second new NodeIds, a selective node renumbering is performed.

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

Command option `-NoOri` asks not to add the `info,OrigNumbering` data in the model stack. `info,OrigNumbering` is only useful when the user needs to convert something specific linked to the new node numerotation that is outside model.

```
% Finding duplicate nodes and merging them
[r1,i2]=feutil('AddNode',model.Node,model.Node);
model=feutil('Renumber',model,r1(i2,1));
```

Renumbering can also be applied to deformation curves, using the same syntax. Be aware however that to keep coherence between a deformation curve and a renumbered model, one should input *`NewNodeNumbers`* as the renumbered model stack entry `info,OrigNumbering`.

```
% Renumering the nodes of a model, and its data
% simple model
model=femesh('testhexa8b');
% simple curve
def=fe_eig(model,[5 5 1e3]);
% first renumber model
model=feutil('renumber',model,1e4);
% then renumber def with renumbering info
r1=stack_get(model,'info','OrigNumbering','get');
def=feutil('renumber',def,r1);
```

RepeatSel *nITE tx ty tz*

Element group translation/duplication. RepeatSel repeats the elements of input model *nITE* times with global axis translations *tx ty tz* between each repetition of the group. If needed, new nodes are added to model.Node. An example is treated in the d_truss demo.

```
% Build a mesh by replicating and moving sub-parts
femesh('Reset'); model=femesh('Testquad4');
model=feutil('Divide 2 3',model);
model=feutil('RepeatSel 3 2 0 0',model); % 3 repetitions, tx=2
feplot(model)
% an alternate call would be
%                                          number, direction
% model=feutil(sprintf('Repeatsel %f %f %f %f', 3,  [2 0 0]))
```

Rev *nDiv OrigID Ang nx ny nz*

Revolution. The elements of model are taken to be the first meridian. Other meridians are created by rotating 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 *x y z* values preceded by an o using a command like model=feutil('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 `model=feutil('Rev 0 101 40 0 0 1',model,[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.

```
% Build a mesh by revolving a sub-part
model=struct('Node',[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],'Elt',[]);
model.Elt=feutil('ObjectBeamLine',1:4);
model=feutil('Divide 3',model);
model=feutil('Rev 40 o 0 0 0 360 0 1 0',model);
feplot(model)
fecom(';triax;view 3;showpatch')
% An alternate calling format would be
%      divi origin angle direct
% r1 = [40  0 0 0  360   0 1 0];
% model=feutil(sprintf('Rev %f o %f %f %f %f %f %f %f',r1))
```

RotateNode *OrigID Ang nx ny nz*

Rotation. The nodes of `model` 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 *x y z* values preceded by an `o` `model=feutil('RotateNode o 2.0 2.0 2.0  90 1 0 0',model)` One can define as a second argument a list of NodeId or a FindNode string command to apply rotation on a selected set of nodes. `model=feutil('RotateNode o 2.0 2.0 2.0  90 1 0 0',model,'x==1')`

For example:

```
% Rotating somes nodes in a model
femesh('reset'); model=femesh('Testquad4'); model=feutil('Divide 2 3',model);
% center is node 1, angle 30, aound axis z
%                                    Center angle  dir
st=sprintf('RotateNode %f %f %f %f %f',[1      30   0 0 1]);
model=feutil(st,model);
feplot(model); fecom(';triax;textnode'); axis on
```

Similar operations can be realized using command `basisgnode`.

SelElt *ElementSelectors*

```
elt=feutil('SelElt ElementSelectors',model)
```

Element selection. SelElt extract selected element from model that verify certain conditions. Available element selection commands are described under the FindElt command and section 7.12 .

SetGroup[*i*,*name*] [Mat *j*, Pro *k*, EGID *e*, Name *s*]

Set properties of a group. For group(s) selected 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

```
% Assigning element properties by groups
model.Elt=feutil('SetGroup1:3 Pro 4',model);
model.Elt=feutil('SetGroup rigid Name celas',model)
```

If you know the column of a set of element rows that you want to modify, calls of the form model.Elt(feutil('FindElt*Selectors*',model),*Column*)= *Value* can also be used. See MPID for higher level custom element properties assignments.

```
% Low level assignment of element properties
femesh('Reset'); model=femesh('Testubeamplot');
model.Elt(feutil('FindElt WithNode{x==-.5}',model),9)=2;
cf=feplot(model);
cf.sel={'groupall','colordatamat'};
```

SetPro, SetMat, GetPro, GetMat

Set an integration property data (ProId) or material property (MatId). You can modify an il or pl property of ID *i* by giving its name and its value using an integrated call of the type

```
% Specifying material/integration rule parameters in a model
 model=femesh('testhexa8');model.il
 model=feutil('SetPro 111 IN=2',model,'MAP',struct('dir',1,'DOF',.01));
 feutilb('_writeil',model)
 mat=feutil('GetPl 100 -struct1',model) % Get Mat 100 as struct
```

The names related to the integration properties a documented in the p_functions, p_solid, p_shell, p_beam, ... To get a type use calls of the form p_pbeam('PropertyUnitTypeCell',1).

The command can also be used to define additional property information : pro.MAP for field at nodes (section **??**), gstate for field at integration points and NLdata for non linear behavior data (nl_spring).

The GetPro and GetMat commands are the pending commands. For example:

```
model=femesh('testhexa8');model.il
rho=feutil('GetMat 100 rho',model) % get volumic mass
integ=feutil('GetPro 111 IN',model) % get the integ rule
```

GetIl, GetPl

The commands GetIl and GetPl respectively output the il and pl matrices of the model for the IDs used by elements. This command provides the values used during assembling procedures and aggregates the values stores in the model.il, model.pl fields and pro, mat entries in the model stack.

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.

Trace2Elt

elt=feutil('Trace2Elt',ldraw);
Convert the ldraw trace line matrix (see ufread 82 for format details) to element matrix with beam1 elements. For example:

```
% Build a beam model from a trace line matrix
TEST.Node=[1001 0 0 0 0 0 0    ; 1003 0 0 0 0.2 0 0 ;
           1007 0 0 0 0.6 0 0  ; 1009 0 0 0 0.8 0 0 ;
           1015 0 0 0 0 0.2 0  ; 1016 0 0 0 0.2 0.2 0;
           1018 0 0 0 0.6 0.2 0; 1019 0 0 0 0.8 0.2 0];
L=[1001 1003 1007 1009];
ldraw(1,[1 82+[1:length(L)]])=[length(L) L];
L=[1015 1016 1018 1019];
ldraw(2,[1 82+[1:length(L)]])=[length(L) L];
L=[1015 1001 0 1016 1003 0 1018 1007 0 1019 1009 0];
```

```
ldraw(3,[1 82+[1:length(L)]])=[length(L) L];
TEST.Elt=feutil('Trace2Elt',ldraw);
cf=feplot(TEST)
```

TransSel *tx ty tz*

Translation of the selected element groups. TransSel replaces elements by their translation of a vector [*tx ty tz*] (in global coordinates). If needed, new nodes are added. Related commands are SymSel, RotateSel and RepeatSel.

```
% Translate and transform a mesh part
femesh('Reset'); model=femesh('Testquad4'); model=feutil('Divide 2 3',model);
model=feutil('TransSel 3 1 0',model); % Translation of [3 1 0]
feplot(model); fecom(';triax;textnode')
```

Please, note that this command is usefull to translate only part of a model. If the full model must be translated, use basiscommand gnode. An example is given below.

```
% Translate all nodes of a model
femesh('Reset'); model=femesh('Testquad4'); model=feutil('Divide 2 3',model);
model.Node=basis('gnode','tx=3;ty=1;tz=0;',model.Node);
feplot(model); fecom(';triax;textnode')
```

UnJoin *Gp1 Gp2*

Duplicate nodes which are common to two element ensembles. To allow the creation of interfaces with partial coupling of nodal degrees of freedom, UnJoin determines which nodes are common to the specified element ensembles.

The command duplicates the common nodes between the specified element ensembles, and changes the node numbers of the second element ensemble to correspond to the duplicate set of nodes. The optional second output argument provides a two column matrix that gives the correspondence between the initial nodes and the duplicate ones. This matrix is coherent with the OrigNumbering matrix format.

The following syntaxes are accepted

- [model,interNodes]=feutil('unjoin *Gp1 Gp2*',model)' Implicit group separation, *Gp1* (resp. *Gp2*) is the group identifier (as integer) of the first (resp. second) element groups to unjoin.
- [model,interNodes]=feutil('unjoin',model,*EltSel1*,*EltSel2*); Separation of two element selections. *EltSel1* (resp. EltSel2) are either FindElt strings or EltId vectors providing the element selections corresponding to each ensemble.

- `[model,interNodes=feutil('unjoin',model,RA);` general input with `RA` as a structure. `RA` has fields
  - `.type`, either *`group`*, *`eltid`* or *`eltind`* that provides the type of data for the selections, set to `eltid` if omitted.
  - `.sel1`, definition of the first element ensemble, the `GroupId` for type group, either a `FindElt` string or a vector of `EltId` or `EltInd` depending on field `.type`.
  - `.sel2`, definition of the second element ensemble, same format as field `.sel1`.
  - `.NodeSel`, provides a `FindNode` selection command to restrict the second element ensemble. Optional, set to `groupall` by default

```
 % Generate a disjointed interface between to parts in a model
femesh('Reset'); model=femesh('Test2bay');
feutil('FindNode group1 & group2',model) % nodes 3 4 are common

% Implicit call for group
mo1=feutil('UnJoin 1 2',model);
feutil('FindNode group1 & group2',mo1) % no common nodes in unjoined model

% Variant by specifying selections
mo1=feutil('UnJoin',model,'group 1','group 2');
feutil('FindNode group1 & group2',mo1) % no common nodes in unjoined model

% Variant with structure input, type "group"
RA=struct('type','group','sel1',1,'sel2',2);
mo1=feutil('UnJoin',model,RA);
feutil('FindNode group1 & group2',mo1) % no common nodes in unjoined model

% Variant with structure input, type "eltid" and string selections
RA=struct('type','eltid','sel1','group1','sel2','group 2');
mo1=feutil('UnJoin',model,RA);
feutil('FindNode group1 & group2',mo1) % no common nodes in unjoined model

% Advanced variants with structure and with selections as vectors
% Clean model EltId
[eltid,model.Elt]=feutil('eltidfix;',model);
i1=feutil('findelt group1',model);
i2=feutil('findelt group2',model);
```

```
% type "eltid"
RA=struct('type','eltid','sel1',eltid(i1),'sel2',eltid(i2));
mo1=feutil('UnJoin',model,RA);
feutil('FindNode group1 & group2',mo1) % no common nodes in unjoined model

% type "eltind"
RA=struct('type','eltind','sel1',i1,'sel2',i2);
mo1=feutil('UnJoin',model,RA);
feutil('FindNode group1 & group2',mo1) % no common nodes in unjoined model
```

See also

feutila, fe_mk, fecom, feplot, section 4.5 , demos gartfe, d_ubeam, beambar ...

feutila

Purpose

Advanced `feutil` commands.

`RotateSel` *`OrigID Ang nx ny nz`*

Rotation. The elements of `model` 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 *x y z* values preceded by an `o`

```
model=feutil('RotateSel o 2.0 2.0 2.0    90 1 0 0',model)
```

Note that old nodes are kept during this process. If one simply want to rotate model nodes, see `RotateNode`.

For example:

```
% Rotate and transform part of a mesh
femesh('reset'); model=femesh('Testquad4');
model=feutil('Divide 2 3',model);
% 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]);
model=feutil(st,model);
feplot(model); fecom(';triax;textnode'); axis on
```

feutilb

Purpose

Gateway function for advanced FEM utilities in SDT.

Description

This function is only used for internal SDT operation and actual implementation will vary over time. The following commands are documented to allow user calls and SDT source code understanding.

AddNode

This command provides optimized operation when compared to the `feutil` equivalent and finer control.

CaseC2SE

Constraint penalization. This command packages the penalization of all constraints in the model. Types `FixDOF`, `RBE3`, `MPC`, `rigid` are supported.

`model=feutilb('CaseC2SE',model,list);` returns the model with penalized constraints. The constraints are then transformed into coupling superelements in the model. *`model`* is an SDT model. *`list`* is an optional restriction cell array of constraint names to be transformed. If omitted all found constraints are penalized.

- `-kp`*`val`* allows defining a custom penalization coefficent. By default the value stored in `sdtdef('kcelas')` is used.
- `-kpAuto` asks to use an automated estimation of kp based on the local compression stiffness in the area concerned by each constraint separately.
- `-keepName` allows keeping the constraints names for superelements. The names are transformed to comply with the superelement naming rules, see section 6.3 for more information. The base case uses names as `typN` with `typ` the type of constraint in lower case and `N` the occurence number in the penalization sequence.
- `-CMT` tells the command to operate on a pre-assembled reduced model `SE,MVR`.

CombineModel

```
mo1=feutilb('combinemodel',mo1,mo2);
[mo1,r1]=feutilb('combinemodel',mo1,mo2);
```

Integrated combining of two separate models. This call aims at creating an assembly from two separate mechanical components. This command properly handles potential `NodeId`, `EltId`, `ProId`, or `MatId` overlaying by setting disjoint ID sets before assembly. Stack or Case entries with overlaying names are resolved, adding `(1)` to common names in the second model. Sets with identical names between both models are concatenated into a single set. The original node numbering matrix for `mo2` is output as a second argument (`r1` in the second example call). The original element numbering matrix for `mo2` can also be output as a third argument.

`mo1` is taken as the reference to which `mo2` will be added, the `Node/Elt` appending is performed by `feutilAddTest`.

- Command option `-cleanMP` cleans up duplicated mat/pro entries in the combined model.
- Command option `-noSetCat`, forces the sets duplication with incremented names (adding `(1)`), instead of concatenation when sets with identical names are found.
- Command option `CompatNodeElt` asks not to shift `NodeId` and `EltId` in the second model. It then assumes the ID ranges are fully compatible in both models.
- Command option `CompatMatPro` asks not to shift `MatId` and `ProId` in the second model. It then assumes these IDs to be fully compatible between both models.
- Command option `CompatBas` asks no to shift the `BasId` in the second model. It then assumes these IDs to be fully compatible between both models.

`GeoLineTopo, ...`

```
r2=feutilb('geolinetopo',model,struct('starts',nodes));
r2=feutilb('geolinetopo',model,struct('starts',RO.nodes(j1,1), ...
      'cos',0,'dir',r1.p(:,2)','circle',r1));
```

`GeoLineTopo` searches a topological line by following mesh edges.

Accepted fields are

- `.starts` node numbers. One row per line.
- `.cos` optional tolerance on direction change used to stop the line.
- `.dir` optional initial search direction, in not provided the direction defined by the line linking the two first nodes is used
- `.forcedir` optional, to force a constant head direction search. This can be used for disturbed lines where local direction variations may induce an unwanted dramatic change or natural direction for the line topology. Quasi-straight lines can thus be obtained in non rules meshes.

- .noSplitTh optional in combination with .forcedir, locally relieves the forcedir constraint if separation of points at a specific step cannot be clearly distinguished along forcedir. In case of non planar topologies, the forcedir head direction may become orthogonal to the local direction seen on the line. In such case, if several points have to be separated for the next line step, one gets the one closer to the forcedir provided. If the forcedir is orthogonal to the currently natural directions, the separation criterion be ill-posed. nlSplitTh provides a tolerance for the dispersion of the next local directions under which the natural local direction is used for the choice instead of the forcedir.
- .circle optional, to use a detection strategy adapted to circle, with richer information. This field is a structure with fields
  - .Origin the coordinates of the circle origin
  - .radius the circle radius
  - .p the local basis associated to the circle principal directions
  - .cos set to zero
  - .dir the normalized direction of the normal to the plane containing the circle.

  This field is mostly defined internally and used by the GeoFindCircle command.

GeoFindCircle packages the GeoLineTopo command to detect nodes on a quasi-circular mesh,

GeoFindCircle, ...

GeoFindCircle searches a topological circular line by following mesh edges. One can either provide three points one the circle, or one point with origin and axis.

```
 r2=feutilb('geofindcircle',model,struct('nodes',[n1 ...]);
 r2=feutilb('geofindcircle',model,...
struct('nodes',n1,'Origin',[x y z],'axis',[nx ny nz]);
```

where n1 is a NodeId, x,y,z are the coordinates of the circle origin, nx, ny, nz is the normal to the plane containing the circle.

The output r2 contains fields

- .Origin the coordinates of the circle origin.
- .normal the normalized direction of the normal to the plane containing the circle.
- .radius the circle radius

- `.p` the local basis associated to the circle principal directions
- `.line` the list of `NodeId` that belong to the circle

The following example illustrates how one can exploit this feature to define a connection screw based on a hole in plates.

```
% use the demostration model for screw defintions with two plates
model=demosdt('demoscrew layer 0 40 20 3 3 layer 0 40 20 4');
% use 3D line pick to find three nodes on the hole
% fe_fmesh('3dlineinit') % acitvate option, and click on 3 nodes on the hole
nodes=[47 43 40]; % nodes picked on the hole
% detect hole
r1=feutilb('geofindcircle',model,struct('nodes',nodes)); r1=r1{1};
n1=feutil('getnodegroupall',model); n2=n1;
% define planes: need to detect plane altitudes
% 1- transform coordinates in the local hole basis for planes generation
n1(:,5:7)=(n1(:,5:7)-ones(size(n1,1),1)*r1.Origin)*r1.p;
[z1,i1]=unique(n1(:,7));
% 2- use global altitudes for the elements detection
z2=n2(i1,7); % use type 1
r2=[num2cell([z1 1+0*z1]) ...
 cellfun(@(x) sprintf('z==%.15g',x),num2cell(z2),'uni',0)];
% 3- screw model, see sdtweb fe_case
r2=struct('Origin',r1.Origin,'axis',r1.normal','radius',r1.radius, ...
 'planes',{r2},...
 'MatProId',[101 101],'rigid',[Inf abs('rigid')],...
 'NewNode',0);
model=fe_caseg('ConnectionScrew',model,'screw1',r2);
% compute modes to test
def=fe_eig(model,[5 10 1e3]);
cf=feplot(model); cf.def=def;
```

GeomRB,[Mass,ByParts,Beam1]

`def=feutilb('geomrb',node,RefXYZ,adof,m)` returns a geometric rigid body modes. If a mass matrix consistent with `adof` is provided the total mass, position of the center of gravity and inertia matrix at CG is computed. You can use `def=feutilb('geomrb cg',Up)` to force computation of rigid body mass properties.

`def=feutilb('geomrbMass',model)` returns the rigid body modes and mass, center of gravity and inertia matrix information. `-bygroup`, `-bymat`, `-bypro` can be used to detail results by subgroups. With no output argument, the results are shown in a table that can be copied to other software.

`def=feutilb('GeomRbByParts',model)` returns the rigid body modes of the model taking into account unconnected regions. Each unconnected mesh region is considered as a different part for which a set of 6 rigid body modes will be generated. `def` containts then a sequence of six rigid body modes by unconnected mesh region, placed in the global model DOF.

`il=feutilb('GeomRBBeam1',mdl,RefXYZ)` returns standard `p_beam` properties for a given model section where `RefXYZ` is the coordinates of the reference point from the gravity center.

`feutilb('GeomRB',mdl,[0 0 0],sens)` or `feutilb('GeomRB',mdl,[0 0 0],Load)` provide a rigid body check of the work generated by loads or loads collocated to sensors on rigid body motion. This provides a direction of application and moments around the origin. These are then used to estimate a point that would lead to the same moments. This point should be on a line of direction of force and containing the actual application point ($x_{true} = x_{est} + \alpha d_x$, ...)

GetUsedNodes

`Node=feutilb('GetUsedNodes',model);` returns the model nodes that are effectively used in the model. This command accounts for nodes present in `SE` elements and nodes used in `Case` constraints that may be not used by elements in `model.Elt`.

```
% Used nodes recovery in a model
% Use a base model with a rigid ring using a node not used by other elements
model=demosdt('demoscrew layer 0 40 20 3 3 layer 0 40 20 4'); % create model
r1=struct('Origin',[20 10 0],'axis',[0 0 1],'radius',3, ...
          'planes',[1.5 0 111 1 3.1;
                    5.0 0 112 1 4;], ...
          'rigid',[Inf abs('rigid')],...
          'NewNode',0);
model=fe_caseg('ConnectionScrew',model,'screw1',r1);
cf=feplot(model); % show model
fecom('promodelviewon');fecom('curtab Cases','screw1');

% Used nodes recovery strategy
n1=feutil('getnodegroupall',cf.mdl); % selects nodes used in model.Elt only
%n2=feutil('optimmodel',cf.mdl); % obsolete call that is based on GetNodeGroupall
n3=feutilb('GetUsedNodes',cf.mdl);

setdiff(n3(:,1),n1(:,1)) % node exclusively used by rigid case
```

Match

Non conform mesh matching utilities. The objective is to return matching elements and local coordinates for a list of nodes.

Matching elements mean

- for volumes, that the physical node is within the element. If volumes may be negative, check orientation using feutil orient.
- for surfaces, that the orthogonal projection of the node is within the element
- for lines that the orthogonal projection on the line is between the line extremities.

A typical node matching call would be

```
% Example of a base match call
model=femesh('test hexa8');
match=struct('Node',[.1 .1 .1;.5 .5 .5;1 1 1]);
match=feutilb('match -info radius .9 tol 1e-8',model,match)

% Example of a matchSurf call
model=demosdt('demoTwoPlate');
% get nodes of half bottom plate
n1=feutil('getnode z==0 & y>.5',model);
% prepare the match structure
match=struct('Node',n1(:,5:7));
% perform surface match on the top plate selection
match=feutilb('matchsurf',model,match,'innode{z==.1}');
% display model and nodes
cf=feplot(model);
fecom(cf,'shownodemark',match.Node,'marker','o'); % display initial nodes
% then overlay matched points
fecom(cf,'shownodemark-noclear',match.StickNode,'marker','s','color','b')

% Use InterpNormal token to get clean normal at matched point
match=struct('Node',n1(:,5:7),'InterpNormal',1);
match=feutilb('matchsurf',model,match,'innode{z==.1}');
fecom(cf,'showmap',struct('vertex',match.StickNode,...
'normal',match.InterpNormal))
```

Accepted command options are

- MatchSurf has the same objective but uses a completely different strategy to match nodes on a surface. This is typically well suited for contact applications (node to surface matching).
  - Note that only the input model skin is treated. This is done by default through a selface command to avoid the need for user treatment for base applications, see FindElt. It is possible to provide in a third argument a FindElt string providing a customized face selection of the model.
  - It is possible to get normals interpolated by shape functions at matched points using the InterpNormal token in the input match structure.
- radius*rad*. The search is limited to points that are not too far a way from matchable element centers. Defining a search radius manually can help prevent matching for elements that are too far away or on the contrary allow matching within elements that are very large so that interior points may be far from the center.
- tol*val* modifies the 1e-8 tolerance used to stop the non-linear search for the match point in second order elements

The output structure contains the fields

.Node	original positions
.rstj	position in element coordinates and jacobian information.
.StickNode	orthogonal projection on element surface if the original node is not within the element, otherwise original position.
.Info	one row per matched node/element giving NodeId if exact match (0 otherwise), number of nodes per element, element type (1 (1D), 2 (2D), 3 (3D), or 5 (SE), an element code and a distance indicator.
.match	obtained when calling the command with -info, typically for row by row post-processing of the match. A cell array with one row per matched node/element giving eltname, slave element row, rstj, sticknode
.slave	an element matrix providing for each node of field .Node the matched element.
.slaveind	the element index (cEGI) in the .Elt matrix of input model providing for each node of field .Node the matched element index.
.master	a sub-index vector providing only the matched nodes in other fields.

MeasThick,Show

Measure of thickness through a volume. Thickness is here defined as the distance from a node on a surface to another surface along the node face normal direction. The base call requires a surface

selection on a wolume mesh from which thickness is measured. The measure in internally performed as a `feutilb Match` call on the other surfaces connected to the surface selected (then assumed fully connected).

The definition of thickness is not unique in the general case, so that peculiar effects can be obtained, especially at corner locations. The defintion chosen here correctly suits thin 3D volumes for which the closest surface nodes to a given surface point is in the opposite surface.

The following commands are supported

- `-sel''EltSel''` can be used to specify a `FindElt` command defining the surface from which the measure is performed.
- `-set''name''` can be used to directly provide a `FaceId` set instead of a seletion through `-sel`.
- `-osel''EltSel''` can be used to restrict match by providing the surfaces facing be the base selection, using a `FindElt` command.
- `-smooth` can be used to smooth the response by interpolating unmatched points or out of tolerance points.
- `-sTol` provides a tolerance over which thickness is considered too large and declared the point unmatched.
- `-show` directly calls command `MeasThickShow` to display the thickness map in `feplot`.

Command `MeasThickShow` performs a display of the thickness map on the mesh in `feplot`.

```
% Thickness measurment and display
model=demosdt('demoUBeam NoPlot'); % demo model
model=feutil('divide 4 4 4',model); % some refinement
[eltid,model.Elt]=feutil('eltidfix;',model); % clean EltId
cf=feplot(model);
RO=struct('sel','selface & innode{y==0.5}',...
 'osel','selface & innode{notin{innode{z==0|z==2.5|y==.5}}}')
d1=feutilb('MeasThick-Show',cf.mdl,RO);
```

MpcFromMatch

This command is used to build multiple point constraints from a match.
`model=feutilb('MpcFromMatch',model,match)`.

The default output is the model with added MPC. The following command options are available

- `-entry` to output the MPC structure instead of the model.
- `-keepAll` not to remove any observation line from the node list.
- `-UseDOF`*`dofi`* to provide alternative DOF, this is usefull for non-mechanical DOF.
- `-UseRot` to keep rotation DOF constraints.
- `Rot` to generate an MPC on rotation DOF only.
- `-NoOff` not to account for StickNode offsets.

The solution retained for surfaces is to first project the arbitrarily located connection point P on the element surface onto a point Q on the neutral fiber used where element nodes are located. Then $Q1$ or $P1$ shape functions and their derivatives are used to define a linear relation between the 6 degree of freedom of point Q and the 3 or 4 nodes of the facing surface. Motion at P is then deduced using a linearized rigid PQ link. One chooses to ignore rotations at the nodes since their use is very dependent on the shell element formulation.

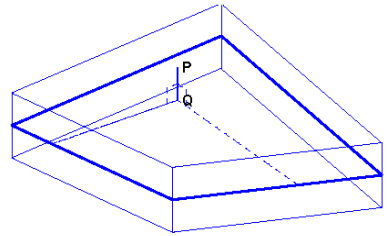


Figure 10.2: Non conform mesh handling

The local element coordinates are defined by $x^e_j, j = 1:3$ along the r coordinate line

$$x^e_j = \alpha_x \frac{\partial N_i}{\partial r} x_{ij} \text{ with } \alpha_x = 1/\left\| \frac{\partial N_i}{\partial r} x_{ij} \right\| \tag{10.3}$$

y^e that is orthogonal to x^e and in the x^e, $\frac{\partial N_i}{\partial s} x_{ij}$ plane, and z^e that defines an orthonormal basis.

The local rotations at point Q are estimated from rotations at the corner nodes using

$$R_j = x^e_j \frac{\partial N_i}{\partial y^e} u_{ik} z^e_k - y^e_j \frac{\partial N_i}{\partial x^e} u_{ik} z^e_k + \frac{1}{2} z^e_j \left(\frac{\partial N_i}{\partial x^e} u_{ik} y^e_k - \frac{\partial N_i}{\partial y^e} u_{ik} x^e_k \right) \tag{10.4}$$

with u_{ik} the translation at element nodes and $j = 1:3, i = 1:N_{node}, k = 1:3$. Displacement at Q is interpolated simply from shape functions, displacement at P is obtained by considering that the segment QP is rigid.

For volumes, displacement is interpolated using shape functions while rotations are obtained by averaging displacement gradients in orthogonal directions

$$\begin{array}{rcl} theta_x & = & \left(-N_{y,z}+Nz,y\right)/2\,\{u\} \\ theta_y & = & \left(N_{x,z}-Nz,x\right)/2\,\{u\} \\ theta_w & = & \left(-N_{x,y}+Ny,x\right)/2\,\{u\} \end{array} \tag{10.5}$$

You can check that the constraints generated do not constrain rigid body motion using `fe_caseg('rbcheck',model)` which builds the transformation associated to linear constraints and returns a list of DOFs where geometric rigid body modes do not coincide with the transformation.

PlaceInDof

This command places DOF based matrices into different sets of DOFs. This can thus be used for `def` curves, observation, constraint, models or matrices. For subsets of DOFs a direct elimination is performed; if the new DOF set contains exclusive DOF, zeros are added, as no expansion is performed here.

This is typically used to eliminate DOFs, add zeros for unused DOFs or simply reorder DOFs. See also `fe_def SubDof`.

High level calls for data structures are supported using syntax
`data= feutilb('PlaceInDof',DOF,data);` where *DOF* is the new set of DOF and *data* is a structure whose fields depends on the type of matrix

- `.def` and `.DOF` are necessary for a deformation field, in coherence with the `def` curve structure.
- `.cta` and `.DOF` for an observation matrix coherent with `sensor` defintions.
- `.c` and `.DOF` for a constraint matrix, coherence with `mpc` definitions. This bears the same base treatment as for the observation matrix but also handles field `.slave` is defined.
- `.K` and `.DOF` for an assembled model. For reduced models the restitution data entry `infoSeRestit` in `.Stack` field is also handled, see `fesuper SEDef` for more information.

The other fields are left unchanged.

Lower level calls for matrices are supported using syntax
`mat=feutilb('PlaceInDof',DOF,oldDOF,mat);`. This call then returns the data matrix placed in the new *DOF* field, assuming that matrix *mat* is based on *oldDOF*. Depending on the size of *mat*, `feutilb` assumes the type of matrix it handles,

- A square matrix of size *oldDOF* is supposed to be a model matrix (stiffness, ...).

- A rectangular matrix with the line dimension equal to the number of *oldDOF* (*i.e.* size(mat,1)==length(oldDOF)) is supposed to be a deformation field.
- A rectangular matrix with the column dimension equal to the number of *oldDOF* *i.e.* size(mat,2)==length(oldDOF)) is supposed to be an observation matrix.

SeparateByMat,Pro

Command SeparateBy ensures that only one MatId or ProId exist in each element group. If a group contains several MatId or ProId the group will be split in the element list, so that the new groups are inserted directly after the currently split group.

By default the criterion is based on MatId, use command SeparateByPro to base it on the ProId.

```
% Separate elements groups by Mat/ProId
% demonstration model
model=demosdt('demoubeam noplot');
% observe element groups
feutil('info',model)
% apply different MatId to different selections
i1=feutil('findelt withnode{z>1&z<=2}',model);
i2=feutil('findelt withnode{z>2}',model);
mpid=feutil('mpid',model.Elt);
mpid(i1,1)=2; mpid(i2,1)=3;
model.Elt=feutil('mpid',model.Elt,mpid);
% now one group with several MatId
feutil('info',model)
% apply group separation
model.Elt=feutilb('SeparateByMat',model.Elt);
% now three groups with unique MatId
feutil('info',model)
```

SubModel

This command aims at extracting a functional model from a selection of an element subset. From a FindElt selection, this command

- Removes unused nodes
- Cleans up the set stack entries. Sets are updated (and removed is they become empty)

- Cleans up the mat/pro entries, unused properties are removed
- Cleans up the case entries, constraints are adapted or removed if external to the submodel (RBE3 or rigid with removed slave or master elements are cleared), loads are adapted or removed if external.
- Updates `info,Rayleigh` and `info,Omega` stack entries.

If the `FindElt` command is omitted or set to `groupall`, the cleanup is performed on the whole model.

The following command options can be used not to clear some specific fields

- `-keepStack` not to clean the stack.
- `-keepCase` not to clean the case stack.
- `-keepMatPro` not to clean pl/il entries.
- `-keepIntSE` to keep superelements whose nodes are fully contained in the selection provided. This option can be usefull to keep coupling superelements when the selection is related to component combinations.

```
% Call to extract a submodel from a model
model=demosdt('demoubeam');
mo1=feutilb('submodel',model,'innode{x<.5}');
feplot(mo1)
```

SurfaceAsQuad[,Group]

This command handles post-treatment of surfaces selections.

The syntax is `mo1=feutilb('SurfaceAsQuad',model,eltsel)'`, where

- `model` is a standard SDT model, that will be transformed
- `eltsel` (optional) is a `FindElt` string that allows a subselection of the initial mesh. The selection should return a face selection, so that the command `& selface` will be added to the `FindElt` string if the token `selface` is missing. If omitted this is set to `selface`.
- `SurfaceAsQuad` command transforms a mesh into a surface `quad4` elements. A mesh surface selection is first performed, triangle surfaces are then transformed intro degenerated `quad4` elements, and second order surfaces are linearized. The output model is then a `quad4` surface mesh.

- SurfaceAsQuadGroup *angle* command splits surfaces based on sharp edges detection. A mesh surface selection is first performed.

  The detection is based on angles between element edges on a surface selection, the threshold is given by *angle* in degrees, if omitted, the default value of 36.87 degrees is taken (corresponding to a cosine value of 0.8). The output model is then a surface mesh divided into groups of surfaces separated by sharp edges. The following command options can be used

  - -set asks not to transform the model, but to generate a meta-set defining the surfaces separated by sharp edges.
  - -set-old asks not to transform the model, but to generate a FaceId set with a connectivity matrix.
  - -isFaceSel asks not to alter the eltsel command even if the token selface is missing. This is useful if one works with a volume based surface selection, not to loose the face identifiers.

The following sample calls illustrate the syntax and the command outputs:

```
% SurfaceAsQuad, transform mesh into quad4 surface
model=femesh('testtetra4'); % sample volume mesh
mo1=feutilb('SurfaceAsQuad',model); % transform into surface quad4 mesh
feutil('info',mo1)

% SurfaceAsQuadGroup, post treat surface selection based on sharp edges
model=femesh('testtetra4'); % sample volume mesh
% Generate the surface mesh with group division by sharp edges
mo1=feutilb('surfaceasquadgroup 90',model);
cf=feplot(mo1); fecom(cf,';colordatagroup;viewn++-;');

% Generate a meta-set named face of FaceId divided by sharp edges
model=feutilb('surfaceasquadgroup90 -set"face"',model);
data=stack_get(model,'set','face','get');
data.SetNames % names of splitted face selections
```

SurfaceSplitDef

This command builds a deformation curve with associated colormap that localizes areas in a model, based on a curve result.

d1=feutilb('SurfaceSplitDef',*model,def,RO*) returns a deformation curve based on def with zeros for non-localized areas and connectivity levels to a starting area. *model* is an SDT model

providing the mesh topology, *def* is a curve based on which areas will be localized and RO is a running option structure with fields

- .elt a boolean telling whether one works with nodes (false) or elements (true).
- .tol that provides a criterion that defined the areas located from initial positions, this is set be default to 0.1.
- .starts that provides a starting point for the area localization. Depending on field .elt this is either a list of nodes or elts, or a string with a command field. Command max*N* is supported and used as a starting list the *N* first maximum values in the curve.
- .sel provides a FindElt string that restricts the initial selection for the clustering.

This command uses the feutilb @levNodeCon object.

```
% SurfaceSplitDef example
% demonstration model
demosdt('demoubeam')
cf=feplot; def=cf.def;
[~,cf.mdl.Elt]=feutil('eltidfix;',cf.mdl);

% Node based field, node clustering
d1=feutilb('surfacesplitdef',cf.mdl,def,struct('tol',.5,'starts','max2'));
cf.def=d1; fecom colordataa
ii_plp('colormap',struct('map',jet(2), ...
 'cval',[0 .01 1],'Band',0,'refine',10,'bSplit',2))

% Element based field
Ek=fe_stress('ener -MatDes 1 -curve',cf.mdl,def);
% Element clustering
d2=feutilb('surfacesplitdef-elt',cf.mdl,Ek,struct('tol',.2,'starts','max2'));
cf.def=def; fecom('colordataelt',d2); colormap(cf.ga,jet);
```

SurfVisible

This command provides visible elements from a particular feplot view.

[eltind,elt,eltindWithHeaders]=feutilb('SurfaceVisible',cf); will output the indices eltind or with headers in cf.mdl.Elt that are currently visible in the display. The second output elt are the face elements consitituting the visible model skin.

This function is compatible but not conforming to `feutil FindElt` command. When outputting elements of different nature than for the model, the base `FindElt` commands will provide empty indices. This function still outputs the visible elements indices of the base model to allow further manipulations.

The following command options are available, either in the input string or in an additionnal running options structure.

- `not` to output invisible elements instead.
- `initsel` to provide an initial selection to peform with `feplot` prior to detection.
- `cv` to provide custom camera postions, as a matrix list of `CameraTarget, CameraPosition, CameraUpVector;...` values. One line per configuration, the output will provide the union of visible elements per view.
- `-rval` to provide a pciture resolution for the dectection algorithm.

```
% Recovering visible elements from a feplot display
model=demosdt('demoubeam-noplot'); % demo model
% tweak its position
model.Node=basis('gnode','rx=45;ry=45;rz=45;',model.Node);
% display in feplot
cf=feplot(model);
% choose a view
fecom(cf','view2');
% Recover visible elements
[ind,elt,i1]=feutilb('SurfVisible',cf);
% restrain view to visible elements
cf.sel='@feutilb(''SurfVisible'',cf)';
```

SurfWjNode

This command provides nodal weights for node based surface integration. The weights are computed as the sum of each element weight contribution using node integration rules.

`r1=feutilb('SurfWjNode',model,sel);`. `model` is a standard SDT model, `sel` is a selection that must provide face elements. If omitted `sel` is set to `selface`, one can provide an empty `sel` is the model is already using shells or the resultof face selection itself. The output `r1` is a structure with fields `.ID` and `.wjdet` respectively providing the `NodeId` and associated surface weights on the given surface.

TKT[,dTKT,TKTSolve]

Optimized matrix projection utilities. This family of commands provides optimized operations obtained through compiled functionalities, and supports out of core, compatible with the sdthdf formats.

- TKT, K = feutilb('tkt',T,K) is the functional equivalent to T'*k*T. K may be a cell array of matrices, in which case one operates on each cell of the array.
- dTKT, r1 = feutilb('dtkt',T,K) is the functional equivalent to diag(T'*k*T) K may be a cell array of matrices, in which case one operates on each cell of the array, the output is then a matrix with the diagonal of each projected matrix on each column.
- TKTSolve, K = feutilb('tkt',T,K,b) is the functional equivalent to T*((T'*k*T) \(T'*b)) that performs a direct resolution with constraints, resolution is called with ofact.

For real bases T, support for RAM footprint optimization is provided through the use of blockwise operations, this can be controlled by the preference BlasBufSize providing a block size in GB. This can be set to Inf to alleviate the behavior. It can be set using sdtdef sdtdef('BlasBufSize',2).

For very large bases T stored in v_handle format through sdthdf command TKTMinRead allows performing blockwise operations on every matrix K at once to limit disc I/O access when loading T. The block sizes are driven by preference OutOfCoreBufferSize providing a memory limit in MB.

Write

feutilb('WriteFileName.m',model) writes a clean output of a model to a script. Without a file name, the script is shown in the command window.

feutilb('_writeil',model) writes properties. feutilb('_writepl',model) writes materials.

Note that this command automatically overwrites existing script files

@levNodeCon

Internal node connectivity object that can be created through its constructor levNodeCon accessed trhough conn=feval(feutilb('@levNodeCon'),[],model);.Note that this call is case sensitive.

The packaged functionalities allow browsing nodes or elts based on element edge levelled connectivity. By default, the node connectivity is initialized only, but one can activate element connectivity with token econ in the construction. conn=feval(feutilb('@levNodeCon'),[],model,'econ');].

Alternative commands allow node/elt expansion based on threholds associated to external data (e.g. an energy curve)

The following methods are available

- expN2Lev. Expands a node list to the node list that is connected up to a given connectivity level. `n2=feval(conn.expN2Lev,conn,[n1; ...]lev);` returns a two column matrix whose column respectively provide the `NodeId` list and the connectivity level from the initial list. `conn` is the connectivity object, `[n1;...]` is a column vector of starting `NodeId`, `lev` is the maximum connectivity level allowed.

- expE2Lev. Expands an element list to the elemnt list that is connected up to a given connectivity level. `elid2=feval(conn.expN2Lev,conn,[elid1; ...]lev);` returns a two column matrix whose column respectively provide the `EltId` list and the connectivity level from the initial list. `conn` is the connectivity object, `[eltid1;...]` is a column vector of starting `EltId`, `lev` is the maximum connectivity level allowed.

- expN2Thr. Generates a node list from a starting list that is incrementally increased by connectivity level so that the final list verifies a given criterion. The criterion is based on a threshold to a quantity to increases with the number of nodes, for example an absolute displacement field or an energy field. `n3=feval(conn.expN2Thr,conn,n1,curve,tol);`. `n1` is the initial list of nodes, `curve` is a data set with fields

  - `.data` is the data field with as many lines as nodes or elements in the model and as many columns as needed.
  - `.ID` or `.EltId` is in coherence with the number of lines of field `.data` and provides either the corresponding `NodeId` for the `.ID` field or `EltId` for the `.EltId` field.
  - `CritFcn` is a criterion function that provides a scalar representative of the value associated to the current node or element list. This is set by default to `crit=feutilb('@scalarCrit')`, the function is called as `val=crit(opt,ind)` with `ind` the indices to be taken on field `.data`. This function should rethrow a positive value increasing with the number of nodes. The list incrementation is stopped once `val>tol`.

- expE2Thr Generates an element list from a starting list that is incrementally increased by connectivity level so that the final list verifies a given criterion. See method expN2Thr for the curve input and criterion function formats.

- getNodes Returns the nodes listed in the `conn` object.

```
% levNodeConn object example node or elt list by connectivity
demosdt('demoubeam')
```

```
cf=feplot; def=cf.def;
% object initialization
conn=feval(feutilb('@levNodeCon'),[],cf.mdl,'econ'); % init
start=1; % eltid 1
eltid=feval(conn.expE2Lev,conn,start,5); % levEltCon
data=struct('EltId',eltid(:,1),'data',eltid(:,2));
fecom('colordataelt',data);
% sample call with ndoes
n2=feval(conn.expN2Lev,conn,[125],2);
```

@unConSel

Internal method whose function handle can be obtained for external use by `unConSel=feutilb('@unConSel');`.Note that this call is case sensitive.

`sel=feval(feutilb('@unConSel'),model);` returns a cell array of `EltId` vectors respectively constituting an unconnected mesh region of the model. The length of the selection is then the number of disconnected mesh regions in the model. This command does not take constraints into account. One has the possibility to work on a model on which constraints have been penalized using command `feutilb CaseC2SE`.

feplot

Purpose

Gateway function for 3-D visualization of structures. See also the companion function `fecom`.

Syntax

```
feplot
feplot(FigHandle)
feplot(model)
feplot(model,def)
```

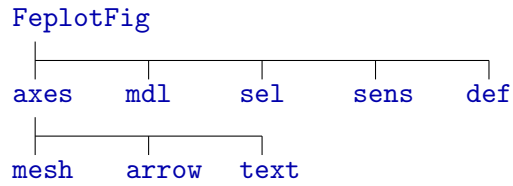
Description

`fecom` gives a complete list of commands. The rest of this section gives more details on the `feplot` architecture. For a tutorial see section 4.4 . Basic ways to call `feplot` are

- `feplot` refreshes all `feplot` axes of the current figure. Use `cf=feplot;cla(cf.ga);feplot` to reinitialze the current plot.
- `cf=feplot` returns a *SDT* `handle` to the current `feplot` figure. You can create more than one `feplot` figure with `cf=feplot(`*`FigHandle`*`)`.
- `cf=cf=comgui('guifeplot -reset -project "SDT Root"',2)` opens an *SDT* `handle` to the specified figure 2. Option `-reset` closes an existing figure if it is not already a `feplot` figure. Option `-project` uses `OsDic` style `Prfeplot` to configure project working directory, image formatting, ...
- `cf=feplot(model)` or `cf.model=model` calls `fecom InitModel` to initialize the FE model displayed in the current figure. See `fecom load` loads the model from a file.
- `cf.def=def` and `cf.def(`*`i`*`)=def` calls `fecom InitDef` to initialize a deformation set.
- `cf=feplot(model,def)` initializes the FE model and a deformation set at the same time.
- `cf.sel={'EltSel','ColorInfo', ... }` calls `fecom Sel` to initialize the selection used to display the model.
- `cf.Stack` and `cf.CStack` calls are detailed in section 4.4.3 .

The old formats `feplot(node,elt,mode,mdof,2)` and `cf.model={Node,Elt}` are 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 `fecom 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.

`mdl` *Model data structure* (see section 7.6) `cf.mdl` is a handle to the model contained in the `feplot` figure. The model must be defined before any plot is possible. It is initialized using the `fecom InitModel` command or using the method `cf.model`.

`Stack` *Model Stack entries* are stored in `cf.mdl.Stack`, but can be more easily reached using `cf.Stack{`*i*`}` or `cf.Stack{`*EntryName*`}` or modified using `cf.Stack{`*EntryType*`,`*EntryName*`}=`*EntryData*.

`CStack` *Case Stack entries* are stored in the stack case (itself stored in `cf.mdl.Stack`). They can be more easily reached using `cf.CStack{`*i*`}` or `cf.CStack{`*EntryName*`}` or modified using `cf.CStack{`*EntryType*`,`*EntryName*`}=`*EntryData*.

`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). `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.

`sens` (obsolete) *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`).`cf.sens(i)={sdof}` initializes a sensor set (see `fecom InitSens`).

`def` *deformation sets* describe deformations at a number of DOFs. Initialized using the `InitDef` command or `cf.def(i)` calls (see `fecom`). `cf.def(i)={def,dof}` is also accepted. `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`).

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 reset 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.

mdl

The model currently displayed is stored in `cf.mdl`, see `fecom InitModel`.

data

The `cf.data` field is used to store volatile interface data. In particular `.ViewClone` can store axes handles that should keep synchronized orientations.

def

The deformations currently displayed are stored in `cf.def`, see `fecom InitDef` for accepted input formats.

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). .ifs gives the element indices (possibly repeated if multiple faces)
.f2	face definitions for lines (if any). .if2 gives the element indices (possibly repeated if multiple faces).
.f1	face definitions for points (if any).
.fvcs	FaceVertexCData for true surfaces (see fecom ColorData commands). Can also be a string, which is then evaluated to obtain the color, or a function handle used in ColorAnimFcn.
.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. The InitSens command is being replaced by the definition of SensDof stack entries.

.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 4.4

fesuper

Purpose

User interface for superelement support.

Syntax

```
              fesuper('CommandString')
[out,out1] = fesuper('CommandString', ...)
model      = fesuper(model,'CommandString', ... )
```

Description

Superelements (see section 6.3 for more details) should be declared as `SE` entries in `model.Stack`, see `fesuper s_` for name restrictions. When using this format, you should specify `model` as the first argument `fesuper` so that any modification to the superelement is returned in the modified stack.

`F ...`

Get full model from superelement model.

```
SE=demosdt('demo ubeam'); SE=SE.GetData; % Load full model.
model=fesuper('SESelAsSe',[],SE); % Build SE model.
Node=fesuper('FNode',model); % Get full model nodes.
Elt=fesuper('FElt',model); % Get full model elements.
mfull=fesuper('FSEModel',model); % Get full model.
```

- `FSEModel` generates a full model (with `.Node` and `.Elt` fields only) based on all SE. **Warning** the output erases the input model, so that care must be taken when model is a `v_handle`. The following command options are available:
  - `-Stack` to keep the initial stack increased with all SE stacks, to keep material properties and sets.
  - `-StackAll` to keep all base stack increased with all SE stacks.
  - `-SESets` to add in the stack element sets corresponding to each SE, under the name `_SE_`*sename*.
  - `-join` to join all element by types in the full model.
- `FElt` outputs the full elements of the model.
- `FNode` outputs the model full nodes. Command `FNodeOptim` outputs the nodes actually used in each SE.

Get,Set ...

Get,set properties from a superelement. Standard superelement fields are detailed in section 6.3.2. `get` and `set` commands are obsolete, you should really use direct access to the `feplot` stack. For example

```
cf=demosdt('demo cmsSE feplot');
SE1=cf.Stack{'se1'};
SE1=stack_set(SE1,'info','EigOpt',[5 10.1 1e3]);
SE1=fe_reduc('CraigBampton -SE -UseDof',SE1);
cf.Stack{'se1'}=SE1; fecom('curtabStack','SE:se1')
```

A new command to perform reduction is under development.
`mdl=fesuper(mdl,'setTR',name,'fe_reduc command')` calls `fe_reduc` to assemble and reduce the superelement. The command option `-drill` can be added to the `fe_reduc` command to consider drilling stiffness in shells. For example `mdl=fesuper(mdl, 'SetTR', 'SE1', 'CraigBampton -UseDof -drill');`
The modes to be kept in the superelement can be set using `mdl=fesuper(mdl, 'setStack', name, 'info', 'EigOpt', EigOptOptions);`

Damp

`model=fesuper('Damp',model,'SEname',damp);` Defines a modal damping on the superelement `SEname`. `damp` can be a scalar `zeta0` and defines a global damping ratio on all computed modes. `damp` can also be a vector `[zeta0 f0 zeta1]` defining a first damping ratio `zeta0` for frequencies lower than `f0` Hz and another damping ratio `zeta1` for higher frequencies. Note that all modes are computed.

SEDef

Superelement restitution. These commands are used to handle model partial or full restitution for visualization and recovery handling.

`SEDefInit` is used to prepare the model for restitution matters. It adds in `model.Stack` an entry `info,SeRestit` containing the necessary data for restitution *i.e.* to perform $\{q\} = [T]\{q_R\}$. This aims to limit generic work needed for multiple restitution. Syntax is
`model=fesuper('SEDefInit',model)`.

`SEDef` is used to implement restitution on full model DOFs. Syntax is `dfull=fesuper ('SeDef', cf, def)`

SEBuildSel

SEBuildSel is used to perform partial restitution on a model. This command sets feplot to display a restitution mesh and computes the corresponding deformation vectors. The restitution selection is defined as a cell array with rows of the form *SeName*,EltSel for selection of each superelement. An EltSel entry set to 'groupall' thus displays the full superelement. EltSel can also be an element matrix (usefull to display deformations on a test frame) or even a vector of NodeIds.
To discard a superelement from display, use an empty string for EltSel. By default a superelement not mentioned in the selection is displayed.

After the generation of superelement selections, it is possible to set a global selection on the full mesh by adding an entry with an empty superelement name (see illustration below).

Accepted command options are

- -nojoin avoids grouping elements of the same topology in a single group.
- -LinFace can be used to generate selections that only use first order faces (tria3 instead of tria6, ...)
- -NoOptim is used to skip the restitution optimization phase.
- -cGL (used in SDT/Rotor) is used in cases with local bases associated with each superelement. In this case, data.cGL is a cell array used to define a local rotation associated with each superelement. Typically, this is equal to data.cGL{jEt}=reshape(mdl.bas(j1,7:15),3,3);.
- -RotDof (used in SDT/Rotor) large angle DOF

The following example is based on a gimbal model reduced in three superelements: base, gimbal and tele. A partial restitution is proposed.

```
model=demosdt('demogimbal-reduce')
cf=feplot(model)
def=fe_eig(model,[5 10 1e3 0 1e-5]);

Sel={'gimbal'  'groupall';
       'tele'  'InNode{z>=0}';
       'base'  ''              };  % base not displayed
fesuper('SEBuildSel',cf,Sel);
cf.def=def;
```

```
% Second selection example
Sel={'gimbal'  'groupall';
       'tele'  '';
       'base'  'groupall'
       '',     'InNode{z>=0}'};  % global selection
fesuper('SEBuildSel',cf,Sel);
```

If you have previously initialized a full restitution with `fesuper('SeDefInit',cf)`, data to optimize partial restitution will be initialized. To obtain a partial restitution of a set of vectors, use `data=cf.sel.cna1;dfull=fesuper('sedef',data,dred)`.

SE ...

SEDof is an internal command used to implement proper responses to `feutil GetDof` commands. It is assumed that the superelement `.DOF` field is defined prior to setting the information in the `model.Stack`.

SEMPC is an internal command that need to be documented.

SECon may also need some documentation.

SEAdd ...

SEAdd*SEName* commands are used to append superelements to a model. With no command option `fesuper('SEAdd SEname',model,SE,[matId proId]'` appends a **new** superelement to the `model.Elt` field (creates a group SE if necessary) and saves the provided SE as a stack entry. `[matId proId]` can be given as a last argument to define properties associated to added superelement. As a new superelement is generated by default, *SEname* can be incremented if a superelement already exists with the same name.

The following command options are available

- `-owrite` allows overwriting a superelement whose name is already assigned.
- `-name''SEname''` can be used instead of letting the superelement name itself in the command, for added robustness.
- `-initcoef` can be used in the case where the superelement is already assembled (reduced part, coupling superelement, ...). This allows the definition of a `p_super`entry of type 2, defining tunable matrix types and coefficients for parametric studies.

- -newID to assign a new independent EltId to each added superelement. This option makes sure that the assigned EltId is not already used in the full model EltId.

Note that *SEname* is checked to comply with the superelement naming convention of SDT, (see section 6.3 , fesuper s_). If *SEname* is altered, a warning will tell how and why. The warning can be deactivated by adding ; at the end of the command string.

SE is usually a standard SDT model, with fields .Node, .Elt, .Stack... But this command accepts models defined only from element matrices (needs .K, .Opt and .DOF fields). It can be useful to cleanly import element matrices from other codes for example (see section 4.3.3), or to represent penalized constraints, see fe_mpc.

When defining a superelement, two node and element numbering coexist, one a the superlement level, and one at the global level. To recover a full model at the global level, see fesuper FSeModel. To control the global model numbering ranges, ones defines NodeId0 and EltId0. NodeId0 is the lower bound of the range of the superelement implicit nodes (use 1 for no shift). NodeIdEnd is given by NodeIdEnd-NodeId0=max(SE.Node(:,1)). EltId0 is the lower bound of the range of the superelement elements. The EltId range width is equal to the maximum EltId of the superelement.

- It is possible to define EltId0 to -1 to let fesuper assign an EltId range over the maximum currently used EltId, accouting for the global model.
- It is possible to define multiple instances of an SE at once (periodic models), see -trans for translation replication and -disk for circular replication. In such case, it is also possible to control the node shift applied between SE by defining a three value series NodeShift NodeId0 EltId0. If only two values are given, NodeShift is defaulted to zero and the two values are interpreted as NodeId0 and EltId0.

SEAdd -unique NodeId0 EltId0 SEname is used to add a single superelement and to give its ranges of implicit nodes and elements.

SEAdd -trans *nrep tx ty tz <NodeShift> NodeId0 EltId0 SEname* is used to repeat the model *nrep* times with a translation step (*tx ty tz*). NodeId0 is the lower bound of the range of the first superelement implicit nodes. The range width is equal to the maximum NodeId of the superelement. The ranges of implicit nodes for repeated superelements are translated so that there is no overlap. To obtain overlap, you must specify *NodeShift NodeId0 EltId0*, then there is a NodeId range overlap of NodeShift nodes. This is used to obtain superelement intersections that are not void and NodeShift is the number of intersection nodes between 2 superelements. EltId0 is the lower bound of the EltId range of elements of the first superelement. There is no EltId range overlap. Option -basval can be used as a starting value for the BasId of superelements.
For example

```
model=femesh('testhexa8');
```

```
model=feutil('renumber',model,model.Node(:,1)*10);
mo1=fesuper('SEAdd -trans 5 0 0 1 10000 10000 cube',[],model)
feplot(mo1)
```

`SEAdd -disk <NodeShift> NodeId0 EltId0 SEName` is used to repeat a sector model in cyclic symmetry. It is assumed that the `symmetry` case entry exists in the model (see `fe_cyclic Build`).

In all these cases, matrix of nodes of the superelement is sorted by NodeId before it is added to the stack of the model (so that `SE.Node(end,1)==max(SE.Node(:,1))`).

`SEAssemble ...`

Command `fesuper('SEAssemble',model)` is used to assemble matrices of superelements that are used in `model`. A basis reduction from superelement `Case.T` (Interface `DofSet` is ignored) is performed.

`SEDispatch ...`

Command `fesuper('SEDispatch',model)` is used to dispatch constraints (`mpc`, `rbe3`, `rigid` elements, ...) of the global model in the related superelements, and create `DofSet` on the interface DOFs.

Rigid elements in `model.Elt` are distributed to the superelements (may be duplicated) that contain the slave node. The master node of the rigid element must be present in the superelement node matrix, even if it is unused by its elements (`SESelAsSE` called with selections automatically adds those nodes to the superelements).

Other constraints (`mpc`, `rbe3`, `FixDof`) are copied to superelement if all constraint DOFs are within the superelement. Constraints that span multiple superelements are not dispatched. All constraints remain declared in the main model. Parameters (`par` entries in Case) are also dispatched if the selection in the superelement is not empty.

Finally a `DofSet` (identity `def` matrix) is defined on superelement DOFs that are active in the global model and shared by another superelement. Those `DofSet` are stored in the `'Interface'` entry of each superelement stack.

`SEDofShow`

Command `fesuper('SeDofShow',cf'tag);` localizes nodes supporting DOF of superelements with mathing name based on *tag* and adds the SE names in an `feplot` display using `cf`. *tag* can be omitted in which case all SE are treated. *tag* can be replaced by a input structure with acceptable fields

- `.tag` provides the tag defined earlier.
- `.evF` provides a function handle to compute custom SE name anchor coordinates from `SE.Node(:,5:7)`. Default uses `@mean`.
- `.sel` provides a custom `feplot` selection command to update display. Default is set to `reset-linface`, use an empty field not to alter the current `feplot` selection.

SEInitCoef ...

Command `fesuper('SEInitCoef',model)` can be used to initialize `p_super` properties in model for used superelements. The full syntax allows choosing the type and a subselection of SE, `[model,pro]=fesuper('SEInitCoef`*`typ`*`',model'`*`sel`*`);`. *typ* can take values `1` or `2` to define the chosen `p_super` type (the default is type 2). *sel* can either be a FindElt string providing SE elements only or a index vector or SE elements in `model.Elt`. The outputs are `model` with additional pro `Stack` entries, and `pro` the list of treated `ProId`.

SEIntNode ...

Command `fesuper('SEIntNode',model)` can be used to define explicitly superelement interface nodes, taking into account local basis.

SESelAsSE ...

Selection as superelement. Command `fesuper('SESelAsSE', model, Sel)` is used to split a model in some superelement models, or to build a model from sub models taken as superelements.
`Sel` can be a FindElt string selector, or a model data structure.
If `Sel` is a FindElt string selector, the elements corresponding to the selection are removed from `model`, and then added as a superelement model. The implicit NodeId of the superelement are the same as the former NodeId in `model`. **Warning:** the selection by element group is not available due to internal renumbering operations performed in this task.
If `Sel` is a model, it is simply added to `model` as a superelement.
`Sel` can also be a cell array of mixed types (FindElt string selector or model data structure): it is the same as calling sequentially a `SESelAsSE` command for each element of the cell array (so avoid using group based selection for example, because after the first selection `model.Elt` may change).
You can give a name to each superelement in the second column of `Sel` `{Selection_or_model,SEname; ...}`. If name is not given (only one column in `Sel`), default `se`*`ID`* is used.
By default, superelements `Mat/ProId` are generated and incremented from `1001`. It is possible to

specify the MatId and/or ProId of the superelements created by adding a third column to Sel, with either a scalar value to apply to MatId and ProId or a line vector under the format [MatId ProId]. *E.g.* Sel={Selection,SEname,[1001 1001];...}. When the third column is left empty for certain lines, the default behavior is applied for these lines only.
Master nodes of the global model rigid elements are added to the superelements that contain corresponding slave nodes. By default, model properties are forwarded to the superelement fields, that is to say il, pl, stack entry types pro, mat, bas, set, and possible stack entries info,Rayleigh and info,Omega.
Superelement addtion is realized with command fesuper SEAdd, additional command options provided in command SeSelAsSe will be forwarded to SEAdd. *E.g.* one can use directly token -newID to generate clean EltId for added superelements.

The following example ▶ divides the d_cms model into 2 sub superelement models.

- The command option -dispatch can be used to dispatch constraints (rigid elements, mpc, rbe3 ...) of the global model in the related superelements and create DofSet on the interface DOFs. It is the same as calling the fesuper SEDispatch command after SESelAsSE without command option.
- Command option -noPropFwd can be used not to forward some model data to the superelement stack (older version compatibility). If used, stack entries of type, pro, mat, bas, set, and possible stack entries info,Rayleigh, info,Omega will not be forwarded to the superelement model.

SERemove

model=fesuper('SERemove',model,'name') searches superelement name in the model and removes it from Stack and element matrix.

SERenumber

SE=fesuper('renumber',model,'name') searches superelement name in the model stack and renumbers based on the entry in the SE element group. If name refers to multiple superelements, you should provide the row number in model.Elt.

s_

Superelement name coding operations. To allow storage in an element row, names must be 8 characters or less, combining letters a...z and numbers 0...9. They are taken to be case insensitive.

For proper use, superelement names should not contain the chain `back`, and should not start with `0`.

`num=fesuper('s_name')` returns the number coding the superelement `name`. `name=fesuper('s_',num)` decodes the number. `elt=fesuper('s_name',model)` extracts elements associated with a given superelement.

See also

`fe_super`, `upcom`, section 4.3.3 , section 6.3

fjlock

Purpose

File lock handling object.

Syntax

```
ob=fjlock(fname); % initialize object
ob.lock(flag); % lock/unlock with flag
state=ob.locked; % lock status
```

Description

To avoid simultaneous file access or to help with keeping track of currently processed files, one can use `fjlock` to test file accessibility or to lock file accessibility to other processes.

`fjlock` behavior follows the following semantics

The lock is handled within the object, ensuring exclusivity even if several `fjlock` objects referring to the same file exist in different processes. The *lock holder* is thus a unique object independently from MATLAB sessions or processes. This leads to three distinct *lock statuses*:

- `0` file unlocked: no external lock found, and not locked in object. The file is accessible but should be locked to safely proceed.
- `1` file externally locked: no access possible, impossible to change the lock status until the lock holder has released it, the process testing accessibility should not proceed.
- `2` file lock within the object: the file is locked but this object is the lock holder, the process using this object may proceed safely.

When a lock holder `fjlock` is destroyed, the lock is released.

`fjlock` inherits the `handle` class, so that any copy refers to the same object (thus same lock status, holder, or destroyed). It is also recommended to use `delete` instead of `clear` to destroy the object. The `clear` command may postpone destruction and thus lock release in recent MATLAB versions.

File locking is never absolutely perfect, as OSes do not use transactional file systems. Besides, POSIX semantics are nowadays weakly enforced to optimize latency, especially over network access, and efficient strategies will depend on the OS. Several lock strategies are thus implemented with their own pros and cons.

- `Java LockFile (flag=1)` This implementation locks the file itself instead of generating a companion lock file. This is a very robust and attractive method but its effect is restrained

within a specific JRE instance (one machine). The lock validity will thus be limited to all MATLAB instances on a specific computer. Once locked the file may become inaccessible to read to any process (even the one hodling the lock). This behavior has been observed on Windows10. A non recoverable segFault may also fail to release the file, that would then remain locked at the JRE level without release access. The JRE would then need to be restarted to recover accessibility.

- `external lock file with Java IO (flag=2,useNIO=0)` This implementation generates a lock file companion whose existence will define the lock status. To be really safe the lock file generation has to be atomic, here through the `File.createNewFile()` method. Specific care is taken to avoid issues related to weak atomicity to the limits of the file system. The lock is then valid with no network limit and cross-platform access. As the file itself is not locked no access issue will exist. This also means that nothing will prevent another rude process to access the file, or delete it, or delete the lock file. The success of this method is thus linked to the robustness of access test. File access recovery in case of object loss is here easily done by deleting the lock file companion.

- `external lock file with Java NIO (flag=2,useNIO=1)` This implementation is a variant to the IO implementation. The main characteristics are the same, but atomicity if realized using the NIO class. The `Files.copy` method is used instead of the `Files.move` method. In recent file system the latter method may be atomic but fails to throw exceptions if the target already exist thus failing the lock scheme. The `Files.copy` associated to an empty file works better, but no atomicity is guaranteed so that a risk of error or lock file corruption may exist, although very small. This method (new from Java7) is eventually limited to recent MATLAB versions. For Unix environments starting from MATLAB 8.2 (R2013b), for Windows environments from MATLAB 9.1 (R2016b).

To be robust to the possibility of several lock strategies used at once, strategy 2 overrides strategy 1 in the lock holder only, and the lock status is independent from the strategy employed. It is then impossible to lock a file if any lock is detected. One can switch in the lock holder from strategy 1 to 2 but not the other way around until the lock is released. Although very improbable, the lock hold could be lost during the switch. It is anyways recommended not to mix strategies within a given distributed procedure.

The default behavior assumes that to be locked, a file must exist. If a file gets deleted, any referred hold lock will be released. It is however sometimes interesting to place a lock on a non-existing file to protect its creation. This specific behavior only works with the external lock files strategies. The operation must then be explicitly called using `flag=3`. In such case the external lock file strategy is forced and a lock can be hold on a non-existing file.

fjlock

`fjlock` constructor. Calling `fjlock` will create a new `fjlock` object. One can provide a file name string *fname*, and a lock *flag* integrer on-the-fly.

```
ob1=fjlock; % create empty object
ob1=fjlock(fname); % refer to file fname, access tested
ob1=fjlock(fname,flag); % refer to file fname and try to lock with flag
```

.delete[,.close]

`fjlock` destruction (and callback). Releases the lock if the object is a lock holder prior to destruction. If no process refers to the object or when exiting MATLAB this method will be called too.

.file

Provide a reference file name. It is possible to change the file reference in an existing object, in such case, the hold locks will be released.

```
ob1.file=fname2; % change fname
```

.lock(*flag*)

Try to assign a lock *flag* to the referred file, and outputs the lock status. If the *flag* is set to false and hold lock is released, otherwise *flag* defines the lock strategy (and not the lock status *per se*). `status=obj.lock(`*flag*`)`.

If the file is locked externally (`status` set to `1`), nothing will be performed, one can however try to lock in a wait loop until the lock hold (`status 2`) is obtained.

Depending on the initial lock status, *flag* setting will have the following effect

- file exists, initial status unlocked (status 0)
  - `flag=0` nothing is done, file remains unlocked.
  - `flag=1` lock with FileLock strategy, lock is hold with strategy 1.
  - `flag=2` lock with external lock file, lock is hold with strategy 2.
  - `flag=3` same as `flag=2`.
- file exists, initial status is externally locked (status 1)

  - `flag=0` nothing is done, file remains externally locked.
  - `flag=1` nothing is done, file remains externally locked.
  - `flag=2` nothing is done, file remains externally locked.
  - `flag=3` nothing is done, file remains externally locked.

- file exists, initial status is lock hold with FileLock strategy (status 2) (stra1)
  - `flag=0` lock is released, file becomes unlocked.
  - `flag=1` nothing is done, lock is hold.
  - `flag=2` switch to external file lock strategy, java FileLock is released, lock is hold.
  - `flag=3` same as `flag=2`.

- file exists, initial status is lock hold with external lock file strategy (status 2) (stra2)
  - `flag=0` lock is released, file becomes unlocked.
  - `flag=1` nothing is done (no strategy change), lock is hold.
  - `flag=2` nothing is done, lock is hold.
  - `flag=3` same as `flag=2`.

- file does not exist, initial status unlocked (status 0)
  - `flag=0` nothing is done, file remains unlocked.
  - `flag=1` nothing is done, file remains unlocked.
  - `flag=2` nothing is done, file remains unlocked.
  - `flag=3` lock with external lock file, lock is hold with strategy 2.

- file does not exist, initial status is lock hold with external lock file strategy (status 2) (stra2)
  - `flag=0` lock is released, file becomes unlocked.
  - `flag=1` nothing is done (no strategy change), lock is hold.
  - `flag=2` nothing is done, lock is hold.
  - `flag=3` nothing is done, lock is hold.

`.locked`

Dynamically provides the object lock status associated to the referred file. Every call to `.locked` thus tests again file accessibility. The output is then the lock status.

```
status=obj.locked;
```

.setFile

Change the referred file in the object. If the object is a lock holder, the lock is released. No lock is performed on the newly referred file.

```
obj.file=fname;
```

.setUseNIO(*flag*)

Change the external file lock implementation strategy. If *flag* is false, the IO strategy will be used, the NIO otherwise. It is recommended to stick with the IO strategy.

.tmpFile

Generate a temporary file using java IO or NIO method. This method is used internally but can also be called externally to generate an empty temporary file with the methods available in Java. This is a variant to nas2up('tempname'), the difference being that .tmpFile directly creates an empty file.

```
f1=char(tmpFile(fjlock,sdtdef('tempdir'),'.mat')); % in tempdir with suffix .mat
f1=char(tmpFile(fjlock,sdtdef('tempdir'))); % in tempdir no suffix
f1=char(tmpFile(fjlock)); % in pwd, no suffix
```

Examples

```
% fjlock calls example
% Generate a file for illustration
f1=char(tmpFile(fjlock,sdtdef('tempdir'),'.mat'));
% Initialize object
ob=fjlock(f1) % dislays .file and .locked
status=ob.locked % 0:  unlocked
% lock the file
status=ob.lock(1) % status is 2
exist([f1 '.fjlock'],'file') % no external file
status=ob.lock(2) % status remains2
exist([f1 '.fjlock'],'file') % external file exists
% unlock the file
ob.lock(0) % status is 0
exist([f1 '.fjlock'],'file') % external file has been removed

% Now try with two objects
```

```
ob1=fjlock(f1) % new lock object
status2=ob1.lock(2) % status2 is 2 ob1 is lock holder
status=ob.locked % status passed to 1 file locked but not by ob
status=ob.lock(0) % status sill to 1 does nothing as ob is not holder
delete(ob1) % lock release when destructed
status=ob.locked % status passed to 0 as no lock exists anymore

f1=nas2up('tempname.mat');
ob=fjlock(f1,2);
status=ob.locked % 0: file does not exist
ob1=fjlock(f1,3);
status1=ob1.locked % 2: lock hold on non existing file
status=ob.locked % 1: external lock found
ob1.lock(0) % 0: lock released
```

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 292). 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

```
% finding DOF indices by extension in a DOF vector
[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

```
% finding DOF indices by NodeId in a DOF vector
 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 200). 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

```
% Generation of observation matrices
 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

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 contains information other than nodes and elements used to describe a FEM computation. Currently supported entries in the case stack are

`cyclic`	(SDT) used to support cyclic symmetry conditions
`DofLoad`	loads defined on DOFs (handled by `fe_load`)
`DofSet`	(SDT) imposed displacements on DOFs
`FixDof`	used to eliminated DOFs specified by the stack data
`FSurf`	surface load defined on element faces (handled by `fe_load`). This will be phased out since surface load elements associated with volume loads entries are more general.
`FVol`	volume loads defined on elements (handled by `fe_load`)
`info`	used to stored non standard entries
`KeepDof`	(obsolete) used to eliminated DOFs not specified by the stack data. These entries are less general than `FixDof` and should be avoided.
`map`	field of normals at nodes
`mpc`	multiple point constraints
`rbe3`	a flavor of MPC that enforce motion of a node a weighted average
`par`	are used to define physical parameters (see `upcom Par` commands)
`rigid`	linear constraints associated with rigid links
`SensDof`	(SDT) Sensor definitions

`fe_case` is called by the user to initialize (when `Case` is not provided as first argument) or modify cases (`Case` is provided).

Accepted commands are

`Get, T, Set, Remove, Reset ...`

- `[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 if it does not exist. 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*.
- `[Case,NNode,ModelDOF]=fe_case(model,'GetT');` returns a congruent transformation matrix which verifies constraints. Details are given in section 7.14 .
  `CaseDof=fe_case(model,'GetTDOF')` returns the case DOF (for model DOF use `feutil('getdof',model)`). If fields `Case.T` and `Case.DOF` are already defined, they will be reused. Use command option `new` to force a reset of these 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')`
- `fe_case SetCurve` has a load reference a curve in model Stack. For example
  `model=fe_case(model,'SetCurve','Point load 1','input');` associates `Point load 1` to curve `input`. See section 7.9 for more details on curves format and `fe_case SetCurve` for details on the input syntax.
- `stack_get` applies the command to the case rather than the model. For example
  `des = fe_case(model,'stack_get','par')`
- `stack_set` applies the command to the case rather than the model. For example
  `model = fe_case(model,'stack_set','info','Value',1)`
- `stack_rm` applies the command to the case rather than the model. For example
  `model = fe_case(model,'stack_rm','par')`

Commands for advanced constraint generation

`AutoSPC`

Analyses the rank of the stiffness matrix at each node and generates a `fixdof` case entry for DOFs found to be singular:

```
model = fe_case(model,'autospc')
```

Assemble

Calls used to assemble the matrices of a model. See `fe_mknl Assemble` and section 4.8.8 for optimized assembly strategies.

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.

ConnectionEqualDOF

`fe_caseg('Connection EqualDOF',model,'name',DOF1,DOF2)` generates a set of `MPC` connecting each DOF of the vector *DOF1* (slaves) to corresponding DOF in *DOF2* (masters). *DOF1* and *DOF2* can be a list of `NodeId`, in that case all corresponding DOF are connected, or only DOF given as a `-dof` *DOFs* command option.

Following example defines 2 disjointed cubes and connects them with a set of `MPC` between DOFs along x and y of the given nodes,

```
% Build a Multiple Point Constraint (MPC) with DOF equalization
% Generate a cube model
cf=feplot; cf.model=femesh('testhexa8');
% duplicate the cube and translate
cf.mdl=feutil('repeatsel 2 0.0 0.0 1.5',cf.mdl);
% build the connection
cf.mdl=fe_caseg('Connection EqualDOF -id7 -dof 1 2',cf.mdl, ...
    'link1',[5:8]',[9:12]');
% display the result in feplot
cf.sel='reset'; % reset feplot display
% open feplot pro and view the built connection
fecom(cf,'promodelviewon');fecom(cf,'curtab Cases','link1');
```

The option `-id` *i* can be added to the command to specify a MPC ID *i* for export to other software. Silent mode is obtained by adding `;` at the end of the command.

By default a DOF input mismatch will generate an error. Command option `-safe` allows DOF mismatch in the input by applying the constraint only to DOF existing in both lists. If no such DOF exists the constraint is not created.

ConnectionPivot

This command generates a set of MPC defining a pivot connection between two sets of nodes. It is meant for use with volume or shell models with no common nodes. For beams the pin flags (columns 9:10 of the element row) are typically more appropriate, see beam1for more details.

The command specifies the DOFs constraint 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.

```
% Build a pivot connection between plates
 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 option `-id i` can be added to the command to specify a MPC ID i for export to other software. Silent mode is obtained by adding `;` at the end of the command.

ConnectionSurface

This command implements node to surface connections trough constraints or penalty.
`fe_caseg('ConnectionSurface DOFs',model,'name',NodeSel1,Eltsel2)` generates a set of MPC connecting of *DOFs* of a set of nodes selected by `NodeSel1` (this is a node selection string) to a surface selected by `EltSel2` (this is an element selection string). `ConnectionSurface` performs a match between two selections using `feutilb Match` and exploits the result with `feutilb MpcFromMatch`.

The following example links x and z translations of two plates

```
% Build a surface connection between two plates
 model=demosdt('demoTwoPlate');
 model=fe_caseg('Connection surface 13 -MaxDist0.1',model,'surface', ...
   'z==0', ...                          % Selection of nodes to connect
   'withnode {z==.1 & y<0.5 & x<0.5}'); % Selection of elements for matching
 def=fe_eig(model);feplot(model,def)
```

Accepted command options are

- `Auto` will run an automated refinement of then provided element selections element selection to locate areas of possible interactions.
- `-aTol` provides a custom tolerance in `Auto` mode to detect interesecting volume extensions where the match will be performed. By default one will consider 10 times the mesh characteristic length.

- `-id i` can be added to the command to specify a MPC ID `i` for export to other software.
- `-Radius val` can be used to increase the search radius for the `feutilb Match` operation.
- `-MaxDist val` eliminates matched node with distance to the matched point within the element higher than `val`. This is typically useful for matches on surfaces where the node can often be external. Using a `-MaxDist` is required for `-Dof`.
- `-kp val` is used to give the stiffness (force/length) for a penalty based implementation of the constraint. The stiffness matrix of the penalized bilateral connection is stored in a superelement with the constraint name.
- `-dens` uses a slave surface. In conjunction with `-kp` the coefficient provided is used as a surface stiffness density. With this option, the first selection must rethrow a face selection.
- `-Dof val` can be used to build surface connections of non structural DOFs (thermal fields, ...).
- `-MatchS` uses a surface based matching strategy that may be significantly faster.
- `-disjCut` will attempt at splitting the generated connection by disjointed connected areas of the surface (second selection), the result is either a series of `mpc` or a model with multiple SE depending on the mode.
- Silent mode is obtained by adding `;` at the end of the command.

It is also possible to define the `ConnectionSurface` implicitly, to let the constraint resolution be performed after full model assembly. The `ConnectionSurface` is then defined as an `MPC`, which `data` structure features fields `.type` equal to `ConnectionSurface` with possible command options, and field `.sel` giving in a cell array a sequence `{NodeSel1, EltSel2}`, as defined in the explicit definition. The following example presents the implicit `ConnectionSurface` definition equivalent to the above explicit one.

```
% Build a surface connection between two plates
% using implicit selections
model=demosdt('demoTwoPlate');
 model=fe_case(model,'mpc','surface',...
struct('type','Connection surface 13 -MaxDist0.1',...
'sel',{{'z==0','withnode {z==.1 & y<0.5 & x<0.5}'}}));
def=fe_eig(model);feplot(model,def)

% Build a penalized surface connection
% with a given sitffness density between two plates
model=demosdt('demoTwoPlate');
```

```
model=fe_caseg('Connection surface 123 -MaxDist 0.1 -kp1e8 -dens',model,...
 'surface',...
 'withnode{z==0}&selface',...
 'withnode {z==.1 & y<0.5 & x<0.5}')
def=fe_eig(model);cf=feplot(model,def);
fecom(cf,'promodelinit');
fecom(cf,'curtabStack','surface');
fecom(cf,'proviewon');
```

Warning volume matching requires that nodes are within the element. To allow exterior nodes, you should add a `& selface` at the end of the element selection string for matching.

ConnectionScrew

```
fe_caseg('Connection Screw',model,'name',data)
```

This command generates a set of RBE3 defining a screw connection. Nodes to be connected are defined in planes from their distance to the axis of the screw. The connected nodes define a master set enforcing the motion of a node taken on the axis of the screw with a set of RBE3 (plane type 1) or rigid links (plane type 0) ring for each plane.

In the case where rigid links are defined, the command appends a group of `rigid` elements to the model case.

Real screws can be represented by beams connecting all the axis slave nodes, this option is activated by adding the field `MatProId` in the `data` structure.

`data` defining the screw is a data structure with following fields:

Origin	a vector [x0 y0 z0] defining the origin of the screw.
axis	a vector [nx ny nz] defining the direction of the screw axis.
radius	defines the radius of the screw.
planes	a matrix with as many lines as link rings. Each row is of the form [z0 type ProId zTol rTol] where z0 is the distance to the origin along the axis of the screw, type is the type of the link (0 for rigid and 1 for rbe3), ProId is the ProId of the elements containing nodes to connect, and zTol and rTol are tolerance of the positions of these nodes respectively for distance along the axis and distance to the axis.
MatProId	Optional. If present beams are added to connect slave nodes at the center of each link ring. It is a vector [MatId ProId] defining the MatId and the ProId of the beams.
MasterCelas	Optional. It defines the celas element which is added if this field is present. It is of the form [0 0 -DofID1 DofID2 ProID EltID Kv Mv Cv Bv]. The first node of the celas is the slave node of the rbe3 ring and the second is added at the same location. This can be useful to reduce a superelement keeping the center of the rings in the interface.
NewNode	Optional. If it is omitted or equal to 1 then a new slave node is added to the model at the centers of the link rings. If it equals to 0, existent model node can be kept.
Nnode	Optional. Gives the number of points to retain in each plane.

Data field planes provide the selection parameters that are used to detect nodes in a plane. The selection combined three FindNode commands in the order detailed below

- nodes are searched in a cylinder using the axis and origin provided, using the radius rTol
- remaining nodes are limited as being over a plane with normal the axis provided, and over position z0 - zTol.
- remaining nodes are limited as begin under a plane with normal the axis provided and under position z0 + zTol.

The found nodes, in a cylinder between two planes of same normal, are then connected to the center node, strictly defined at height z0 on the axis provided. The heights provided as z0 and zTol must be understood along the axis provided and not as function of the main frame coordinates.

One can also define more generally planes as a cell array whose each row defines a plane and is of the form {z0 type st} where z0 and type are defined above and st is a FindNode string. st can contain $FieldName tokens that will be replaced by corresponding data.FieldName value (for example 'cyl<= $radius o $Origin $axis & inElt{ProId $ProId}' will select nodes in cylin-

der of radius data.radius, origin data.Origin and axis data.axis, and in elements of ProId data.ProId).

Silent mode is obtained by adding ; at the end of the command.

Following example creates a test model, and adds 2 rbe3 rings in 2 planes.

```
% Sample connection builds commands for screws using rigid or RBE3
model=demosdt('demoscrew layer 0 40 20 3 3 layer 0 40 20 4'); % create model
r1=struct('Origin',[20 10 0],'axis',[0 0 1],'radius',3, ...
          'planes',[1.5 1 111 1 3.1;
                    5.0 1 112 1 4;], ...
          'MasterCelas',[0 0 -123456 123456 10 0 1e14], ...
          'NewNode',0);
model=fe_caseg('ConnectionScrew',model,'screw1',r1);
cf=feplot(model); % show model
fecom('promodelviewon');fecom('curtab Cases','screw1');

% alternative definintion using a beam
model=demosdt('demoscrew layer 0 40 20 3 3 layer 0 40 20 4'); % create model
r1=struct('Origin',[20 10 0],'axis',[0 0 1],'radius',3, ...
          'planes',[1.5 1 111 1 3.1;
                    5.0 1 112 1 4;], ...
          'MasterCelas',[0 0 -123456 123456 10 0 1e14], ...
          'MatProId',[110 1001],...
          'NewNode',0);
model=fe_caseg('ConnectionScrew',model,'screw1',r1);
cf=feplot(model); % show model
fecom('promodelviewon');fecom('curtab Cases','screw1');

% alternative definition with a load, two beam elements are created
model=demosdt('demoscrew layer 0 40 20 3 3 layer 0 40 20 4'); % create model
model=fe_caseg('ConnectionScrew -load1e5;',model,'screw1',r1);
def=fe_eig(model,[5 15 1e3]);

% alternative definition with a load, two beam elements are created
% and a pin flag is added to release the beam compression
model=demosdt('demoscrew layer 0 40 20 3 3 layer 0 40 20 4'); % create model
model=fe_caseg('ConnectionScrew -load1e5 -pin1;',model,'screw1',r1);
def1=fe_eig(model,[5 15 1e3]);
```

```
% a new rigid body mode has been added due to the pin flag addition
[def.data(7) def1.data(7)]
```

Command option -load*val* allows defining a loading force of amplitude *val* to the screw in the case where a beam is added to model the screw (through the MatId optional field). To this mean the last beam element (in the order defined by the planes entry) is split in two at a tenth of its length and a compression force is added to the larger element that is exclusively inside the beam. In complement, command option -pin*pdof* allows defining pin flags with identifiers *pdof* to the compressed beam1element.

Entries

The following paragraphs list available entries not handled by fe_load or upcom.

cyclic (SDT)

cyclic entries are used to define sector edges for cyclic symmetry computations. They are generated using the fe_cyclic Build command.

FixDof

FixDof entries correspond to rows of the Case.Stack cell array giving {'FixDof', Name, Data}. Name is a string identifying the entry. data is a column DOF definition vector (see section 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 DOF and ID specifications to the findnode command. For example 'x==0 -dof 1 2 -ID 101' fixes DOFs x and y on the x==0 plane and generates an data.ID field equal to 101 (for use in other software).

The following command gives syntax examples. An example is given at the end of the fe_case documentation.

```
% Declare a clamping constraint with fixdof
 model = fe_case(model,'FixDof','clamped dofs','z==0', ...
    'FixDof','SimpleSupport','x==1 & y==1 -DOF 3', ...
    'FixDof','DofList',[1.01;2.01;2.02], ...
    'FixDof','AllDofAtNode',[5;6], ...
    'FixDof','DofAtAllNode',[.05]);
```

map

`map` entries are used to define maps for normals at nodes. These entries are typically used by shell elements or by meshing tools. `Data` is a structure with fields

- `.normal` a N by 3 matrix giving the normal at each node or element
- `.ID` a N by 1 vector giving identifiers. For normals at integration points, element coordinates can be given as two or three additional columns.
- `.opt` an option vector. `opt(1)` gives the type of map (1 for normals at element centers, 2 for normals at nodes, 3 normals at integration points specified as additional columns of `Data.ID`).
- `.vertex` an optional N by 3 matrix giving the location of each vector specified in `.normal`. This can be used for plotting.

MPC

`MPC` (multiple point constraint) entries are rows of the `Case.Stack` cell array giving `{'MPC', Name, Data}`. `Name` is a string identifying the entry. `Data` is a structure with fields `Data.ID` positive integer for identification. `Data.c` is a sparse matrix whose columns correspond to DOFs in `Data.DOF`. `c` is the constraint matrix such that $[c]\{q\} = \{0\}$ for q defined on `DOF`.

`Data.slave` is an optional vector of slave DOFs in `Data.DOF`. If the vector does not exist, it is filled by `feutil FixMpcMaster`.

Note that the current implementation has no provision for using local coordinates in the definition of MPC (they are assumed to be defined using global coordinates).

par (SDT)

`par` entries are used to define variable coefficients in element selections. It is nominally used through `upcom Par` commands but other routines may also use it [31].

RBE3 (SDT)

`rbe3` constraints enforce the motion of a slave node as a weighted average of master nodes. Two definition strategies are supported in SDT, either direct or implicit. There are known robustness problems with the current implementation of this constraint.

The direct definition explicitely declares each node with coupled DOFs and weighting in a `data` field. Several `rbe3` constrains can be declared in `data.data`. Each row of `data.data` codes a set of constraints following the format

`Rbe3ID NodeIdSlave DofSlave Weight1 DofMaster1 NodeId1 Weight2 ...`

`DofMaster` and `DofSlave` code which DOFs are used (123 for translations, 123456 for both translations and rotations). You can obtain the expression of the RBE3 as a MPC constraint using `data=fe_mpc('rbe3c',model,'CaseEntryName')`.

When reading NASTRAN models an alternate definition

`Rbe3ID NodeIdSlave DofSlave Weight DofMaster NodeId1 NodeId2 ...` may exist. If the automated attempt to detect this format fails you can fix the entry using `model=fe_mpc('FixRbe3Alt',model)`.

The implicit definition handles *Node Selectors* described in section 7.11 to define the `rbe3`. The input is then a structure:

```
% Define a RBE3 constraint
data=struct('SlaveSel','NodeSel',...
            'MasterSel','NodeSel',...
            'DOF', DofSlave,...
            'MasterDOF', DofMaster);
```

`SlaveSel` is the slave node selection (typically a single node), `MasterSel` is the master node selection, `DOF` is the declaration of the slave node coupling, `MasterDOF` is the declaration of the master nodes coupling (same for all master nodes).

Grounding or coupling the slave node movement is possible through the use of a celas, as shown in the example below featuring an implicit `rbe3` definition. In a practical approach, the slave node is duplicated and a `celas` element is generated between the two, which allows the definition of global movement stiffnesses. Constraining the rotation of a drilled block around its bore axis is considered using a global rotation stiffness.

```
% Integrated generation of an RBE3 constraint in a model
% Definition of a drilled block around y
model=feutil('ObjectHoleInBlock 0 0 0   1 0 0   0 1 0  2 2 2 .5 4 4 4');
model=fe_mat('DefaultIl',model); % default material properties
model=fe_mat('defaultPl',model); % default element integration properties
% Generation of the bore surface node set
[i1,r1]=feutil('Findnode cyl ==0.5 o 0 0 0 0 1 0',model);
model=feutil('AddsetNodeId',model,'bolt',r1(:,1));
% Generation of the slave node driving the global bore movement
```

```
model.Node(end+[1:2],1:7)=[242 0 0 0 0 0 0;244 0 0 0  0 0 0];
% Addition of the celas element between the slave node and its duplicate
model.Elt(end+[1:2],1:7)=[inf abs('celas') 0;242 244 123456 0 0 0 1e11];
model=feutil('AddSetNodeId',model,'ref_rot',244);
% Definition of the RBE3 constraint
data=struct('SlaveSel','setname ref_rot',...
            'MasterSel','setname bolt',...
            'DOF',123456,... % Slave node constrained on 6 DOF
            'MasterDOF',123); % Master only use translation
model=fe_case(model,'rbe3','block_mov',data);
% Grounding the global y rotation (leaving the celas stiffness work)
model=fe_case(model,'fixdof','ClampBlockRot',242.05);
% 5 rigid body modes model obtained
def=fe_eig(model,[5 20 1e3]);
cf=feplot(model,def);fecom('curtabCases','rbe3');fecom('ProViewOn');
```

rigid

See details under `rigid` which also illustrates the `RigidAppend` command.

Sens ... (SDT)

`SensDof` entries are detailed in section 4.6 . They are stored as rows of the `Case.Stack` cell array giving `{'SensDof', Name, data}`. `SensStrain` entries have been replaced with strain sensors in `SensDof`.

`R1=fe_case('sensobserve',model,'SensEntryName',def); iiplot(R1)` can be used to extract observations at sensors associated with a given response. The `SensEntryName` can be omitted if a single sensor set exist.
`Sens=fe_case(model,'sens','SensName');R1=fe_case('sensobserve',Sens,def);` is also acceptable

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.

SetCurve

To associate a time variation to a compatible case entry, one adds a field `curve` to the case entry structure. This field is a cell array that is of the same length as the number of solicitation contained in the case entry.

Each curve definition in the cell array can be defined as either

- a string referring to the name of a curve stacked in the model (recommended)
- a curve structure
- a string that will be interpreted on the fly by `fe_curve`when the load is assembled, see `fe_curve('TestList')` to get the corresponding strings

The assignation is performed using

```
model = fe_case(model,'SetCurve',EntryName,CurveName,Curve,ind);
```

with

- `EntryName` the case entry to which the curve will be assigned
- `CurveName` a string or a cell array of string with the name of the curves to assign
- `Curve` (optional) a curve or a cell array of curves that will be assigned (if not in model stack), they will be set in the model stack and only their names will be mentioned in the case entry
- `ind` (optional) the index of the curves to assign in the `curve` field, if several solicitation are present in the case entry considered. If `ind` is omitted the whole field `curve` of the case entry will be replaced by `CurveName`.

In practice, a variant call is supported for retro-compatibility but is not recommended for use,

```
model = fe_case(model,'SetCurve',EntryName,Curve,ind);
```

allows a direct assignation of non stacked curves to the case entry with the same behavior than for the classical way.

Multiple curve assignation at once to a specific `EntryName` is supported with the following rules

- `CurveName`, `Curve` (optional) and `ind` (mandatory) have the same sizes. In this case, all given curves will be assigned to the case entry with their provided index

- A singleCurveName and Curve is provided with a vector of indices. In this case, all indexed curves will be assigned to the new provided one

To remove a curve assignation to a case entry. Command

```
model = fe_case(model,'SetCurve',EntryName,'remove');
```

will remove the field curve from case entry EntryName.

The flexibility of the command imposes some restriction to the curve names. Name remove and TestVal with Val begin a keyword used by fe_curve Test cannot be used.

The following example illustrate the use of SetCurve to assign curves to case entries

```
% Sample calls to assign curves to load cases
% generate a sample cube model
 model=femesh('testhexa8');
 % clamp the cube bottom
 model=fe_case(model,'FixDof','clamped dofs','z==0');
 % load a DOF of the cube base
 model=fe_case(model,'DofLoad','in',struct('def',1,'DOF',5.02));
 % generate a curve loading transient pattern
 R1=fe_curve('testramp t1.005 yf1');
 % assign the curve to the load case
 model=fe_case(model,'SetCurve','in','tramp',R1);

 % add a new load case with two sollicitations
 model=fe_case(model,'DofLoad','in2',...
  struct('def',[1 0;0 1],'DOF',[6.02;6.03]));
 % assign a new transient variation to both directions
 model=fe_case(model,'SetCurve','in2','tramp1',...
  fe_curve('testramp t0.5 yf1'),1:2);
 % modify the first direction only to tramp instead of tramp1
 model=fe_case(model,'SetCurve','in2','tramp',1);

 % remove the curve assigned to input in
 model=fe_case(model,'SetCurve','in','remove')
```

Examples

Here is an example combining various fe_case commands

```
% Sample fe_case commands for boundary conditions, connections, and loads
femesh('reset');
model = femesh('test ubeam plot');
% specifying clamped dofs (FixDof)
model = fe_case(model,'FixDof','clamped dofs','z==0');
% creating a volume load
data  = struct('sel','GroupAll','dir',[1 0 0]);
model = fe_case(model,'FVol','Volumic load',data);
% assemble active DOFs and matrices
model=fe_mknl(model);
% assemble RHS (volumic load)
Load  = fe_load(model,'Case1');
% compute static response
kd=ofact(model.K{2});def.def= kd\Load.def; ofact('clear',kd)
Case=fe_case(model,'gett'); def.DOF=Case.DOF;
% plot displacements
feplot('initdef',def);
fecom(';undef;triax;showpatch;promodelinit');
```

See also fe_mk, fe_case

fe_caseg

Purpose

Gateway functions for advanced FEM utilities in SDT, regarding assembly, integrated case definition and post-treatments.

Description

This function is only used for internal SDT operation and actual implementation will vary over time. The following commands are documented to allow user calls and SDT source code understanding.

Assemble

Optimized strategies for assembly are provided in SDT through the `fe_caseg Assemble` command. More details are given in section 4.8.8 .

StressCut

The `StressCut` command is the gateway for dynamic stress observation commands. Typical steps of this command are

- View mesh generation, see section 4.7.1 .
- Generate a selection `sel=fe_caseg('stresscut -selout',VIEW,model);`
- Display the selection in `feplot` using `fe_caseg('stresscut',sel,cf)`
- Observe the result using `curve=fe_caseg('StressObserve',cf.sel(2),def)`

For the selection generation, accepted options are

- `VIEW` can be a mesh so that `feutilb Match` is used to find elements associated with viewing positions. A structure `struct('type','Gauss')` to return selection at Gauss points. A structure `struct('type','BeamGauss')` to return selection at beam Gauss points.
- a `model` or `feplot` handle `cf` can be provided as third argument.
- `-SelOut` requires selection output.
- `-Radius`*`val`* provides a search radius for the `feutilb Match` call.

The sel data structure is a standard selection (see feplot sel) with additional field .StressObs a structure with the following fields

- .cta observation matrix for stress components. The expected sort is to have all components at first node, all at second node, ...
- .DOF expected DOF needed for the observation.
- .X,.Xlab labels for the observation, see Multi-dim curve for details.
- .CritFcn callback to be evaluated, see fe_stress CritFcn.
- .Node,.Elt nodes and elements for the view mesh.
- .trans structure for the observation of interpolated displacement (needed when view mesh nodes are not nodes of the original mesh).

Par[Mat,Pro,SE,Init,Set,2Case]

Advanced parameter declaration in models. Lower level declaration can be found in fe_case par. Model parametrization framework can be found in XXX.

The following commands are available to declare SDT parameters

- ParMat Support to declare as parameter and possibly split a material property. **Warning:** Some formulations and parameter classes cannot directly be split from the constitutive law, in such case the resulting assembled matrices may not be computable. Advances material splitting features are available in the Viscoelastic toolbox [31]. Syntax is
  `model=fe_caseg('ParMat',model,'p1 .... -matid i',par);` with model a SDT model, *p1* is a constitutive law parameter as declared in the corresponding m_ function, and par is a parameter entry. The working material is defined by the token -matid. The output model can have a split material featuring varying parameters, and will have a Case par entry declaring the parameter and a entry in Stack,Range0 providing its variation.
- ParPro Support to declare as parameter and possibly split an integration property, this is designed for discrete structural elements such as celas, cbush, mass1elements. Syntax is
  `model=fe_caseg('ParPro',model,'p1 .... -proid i',par);` with model a SDT model, *p1* is a constitutive law parameter as declared in the corresponding p_ function, and par is a parameter entry. The working property is defined by the token -proid. The output model can have duplicated elements featuring varying parameters, and will have a Case par entry declaring the parameter and a entry in Stack,Range0 providing its variation.

- ParSE Support to declare as parameter a superelement, or a subset of superelement matrices. One can identify the SE of interest either by its SeName or its ProId. If necessary one can identify matrices of interest either by Klab or matdes, or property name in the p_superentry.
  ```
  model=fe_caseg('ParSE',model,' -SeName"se1"',par);
  model=fe_caseg('ParSE',model,'coef1 -proid1001',par);
  model=fe_caseg('ParSE',model,' matdes3 -proid1001',par);
  ```
  The output model can have duplicated elements featuring varying parameters, and will have a Case par entry declaring the parameter and a entry in Stack,Range0 providing its variation.

The following commands are available to declare and handle broader parameter definitions, to be used in dedicated routines

- ParInit Instantiate .param entries in supported model features.
  model=fe_caseg('ParInit',model,par); par is here a parameter or a cell array of parameters to be implemented. Implementation or the feature to be affected is provided through the .info field of the parameter. It is a string following the format type>entry TokenId.
  - type is optionnal (> is then omitted) and provides a way of defining field .type of the parameter usual types are double, pop, but other custom types can be defined for dedicated applications.
  - entry defines the parameter effect, the value depends on the type of feature to be parametered, defined by the TokenId
  - TokenId defines the feature on which the parameter is applied. The following features are supported
    * Materials, either defined by -mat*name* or -matid*i*. Acceptable entries as then any declared constitutive law in the corresponding m_ function.
    * Structural properties, either defined by -pro*name* or -proid*i*. Acceptable entries as then any declared constitutive law in the corresponding p_ function. Properties in NLdata are supported, in such case the entry must start with nldata.*val* to affect field *.val* of field .NLdata.
    * Loads, defined by their type and name type*name* (e.g. -dofLoad''ExForce''). entry is then the impacted field name.
    * Boundary conditions, defined by their type and name type*name* (e.g. -rigid''conn''). entry is then the impacted field name.

- ParSet Applies a current parameter set (or design point) to a model for which fe_caseg ParInit has been applied. Given an SDT model and a Range structure with field .jPar, the procedure loops over supported features having a .param field, and applies the current values. model=fe_caseg('ParSet',model,Range);.

- Par2Case XXX Loads and ParPro at the moment

StressObserve

The StressCut command typically returns all stress components (x, y, and z), for a relevant plot, it is useful to define a further post-treatment, using the sel.StressObs.CritFcn callback. This callback is called once the stress observation have been performed. The current result is stored in variable r1, and follows the dimensions declared in field .X of the observation. For example to extract stresses in the x direction, the callback is

```
sel.StressObs.CritFcn='r1=r1(1,:,:);';
```

The StressObserve command outputs the stress observation in an curve structure. You can provide a callback -crit "my_callback". The command option -trans allows observation of translations for selections that have this observation. If empty, all components are kept.

```
data=fe_caseg('StressObserve -crit""',cf.sel(2),def);
iiplot(data); % plot results
```

ZoomClip

The command accessible through the axes context menu Clip, can now also be called from the command line fe_caseg('ZoomClip',cf.ga,[xyz_left;xyz_right]).

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},ceigopt)
        ...  = fe_ceig({m,c,k,T,mdof},ceigopt)
        ...  = fe_ceig(model,ceigopt)
        ...  = fe_ceig( ... ,flag)
```

Description

Complex modes are solution of the second order eigenvalue problem (see section 5.5 for details)

$$[M]_{N\times N}\{\psi_j\}_{N\times 1}\lambda_j^2+[C]\{\psi_j\}\lambda_j+[K]\{\psi_j\}=0 \tag{10.6}$$

where modeshapes `psi`=ψ and poles $\Lambda=\left[\backslash\lambda_{j\backslash}\right]$ are also solution of the first order eigenvalue problem (used in `fe_ceig`)

$$\left[\begin{array}{cc} C & M \\ M & 0 \end{array}\right]_{2N\times 2N}\left[\begin{array}{c} \psi \\ \psi\Lambda \end{array}\right]_{2N\times 2N}[\Lambda]_{2N\times 2N}+\left[\begin{array}{cc} K & 0 \\ 0 & -M \end{array}\right]\left[\begin{array}{c} \psi \\ \psi\Lambda \end{array}\right]=[0]_{2N\times 2N} \tag{10.7}$$

and verify the two orthogonality conditions

$$\psi^T C\psi+\Lambda\psi^T M\psi+\psi^T M\psi\Lambda=I \quad\text{and}\quad \psi^T K\psi-\Lambda\psi^T M\psi\Lambda=-\Lambda \tag{10.8}$$

If matrices are non-symmetric, the left eigenvectors differ from the right eigenvectors. One can then set input `flag` to *'lr'* to obtain the left eigenmodes in the output `def` structure. See section 7.8 to get more information about the `def` structure.

`[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 `EigOpt` are standard `fe_eig` options and `CeigMethod` can be

- 0 (full matrices)
- 1 real modes then complex ones on the same basis (equivalent to NASTRAN SOL 110)
- 2 real modes and first order correction for viscous and hysteretic damping part.
- 3 is a refined solvers available with the VISCO extension.

Here is a simple example of `fe_ceig` calls.

```
model=demosdt('demoubeam'); cf=feplot;
[Case,model.DOF]=fe_mknl('init',model);
m=fe_mknl('assemble not',model,Case,2);
k=fe_mknl('assemble not',model,Case,1);

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,model.DOF},[1 6 10 1e3]); % fixed modes

cf.def=def1; % show def1 in feplot figure
```

See also

`fe_eig`, `fe_mk`, `nor2ss`, `nor2xf`, section 5.3

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`

$$\text{range}([T]_{N\times(N-NC)}) = \ker([c]_{NS\times N}) \tag{10.9}$$

$NC \leq NS$ 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

$$T = \left[[T_U]_{N\times NS}\,[T_K]_{N\times(N-NS)}\right] \text{ with } [c]_{NS\times N}\,[T] = \left[\begin{bmatrix}\backslash\\ I\\ \backslash\end{bmatrix}\,[0]_{NS\times(N-NS)}\right] \tag{10.10}$$

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

$$\begin{array}{c}[Ms^2+Cs+K]\,\{q(s)\} = [b]\,\{u(s)\}\\ \{y(s)\} = [c]\,\{q(s)\}\end{array} \quad \text{with } [c_{int}]\,\{q(s)\} = 0 \tag{10.11}$$

which are often found in CMS problems (see section 6.2.6 and [40]).

To eliminate the constraint, one determines a basis T for the kernel of $[c_{int}]$ and projects the model

$$\begin{gathered}\left[T^T M T s^2 + T^T C T s + T^T K T\right]\{q_R(s)\} = \left[T^T b\right]\{u(s)\} \\ \{y(s)\} = [cT]\{q_R(s)\}\end{gathered} \tag{10.12}$$

See also

Section 7.14, `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 . The iiplot interface may be used to plot curves and a basic call would be iiplot(Curve) to plot curve data structure Curve.

Accepted commands are

bandpass *Unit f_min f_max*

out=fe_curve('BandPass *Unit f_min f_max*',signals);
realizes a true bandpass filtering (i.e. using fft() and ifft()) of time signals contained in curves signals. *f_min* and *f_max* are given in units *Unit*, whether Hertz(Hz) or Radian(Rd). With no *Unit*, f_min and f_max are assumed to be in Hertz.

```
% apply a true bandpasss filter to a signal
out=fe_curve('TestFrame');% 3 DOF oscillator response to noisy input
fe_curve('Plot',out{2});  % "unfiltered" response
filt_disp=fe_curve('BandPass Hz 70 90',out{2}); % filtering
fe_curve('Plot',filt_disp); title('filtered displacement');
```

datatype [,cell]

out=fe_curve('DataType',DesiredType);
returns a data structure describing the data type, useful 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. One can specify the unit with out=fe_curve('DataType',DesiredType,'unit');.

DataTypeCell returns a cell array rather than data structure to follow the specification for curve data structures. Command option -label"*lab*" allows directly providing a custom label named *lab* in the data type.

getcurve

```
curve=fe_curve('getcurve',model,'curve_name');
```
extracts curve `curve_name` from `model.Stack` or the possible curves attached to a load case. If the user does not specify any name, all the curves are returned in a cell array.

h1h2 *input_channels*

```
FRF=fe_curve('H1H2 input_channels',frames,'window');
```
computes H1 and H2 FRF estimators along with the coherence from time signals contained in cell array `frames` using window `window`. The time vector is given in `frames{1}.X` while *`input_channels`* tells which columns of in `frames{1}.Y` are inputs. If more than one input channel is specified, true MIMO FRF estimation is done, and Hν is used instead of H2. When multiple frames are given, a mean estimation of FRF is computed.

Note: To ensure the proper assembly of H1 and Hν in MIMO FRF estimation case, a weighing based on maximum time signals amplitude is used. To use your own, use
`FRF=fe_curve('H1H2 input_channels',frames,window,weighing);`
where `weighing` is a vector containing weighing factors for each channel. To avoid weighing, use
`FRF=fe_curve('H1H2 input_channels',frames,window,0);` . For an example see
`sdtweb('start_time2frf','h1h2')`

noise

OBSOLETE : use `fe_curve TestNoise` instead

```
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 until `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`.

```
% computes 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);
noise_f=fe_curve('fft',noise_h);
figure(1);
subplot(211);fe_curve('plot -gca',noise_h);axis tight;
subplot(212);fe_curve('plot -gca',noise_f);axis tight;
```

plot

`fe_curve('plot',curve);` plots the curve `curve`.
`fe_curve('plot',fig_handle,curve);` plots `curve` in the figure with handle `fig_handle`.
`fe_curve('plot',model,'curve_name');` plots the curve of `model.Stack` named `curve_name`.
`fe_curve('plot',fig_handle,model,curve_name);` plots curve named `curve_name` stacked in `.Stack` field of model `model`.

```
% Plot a fe_curve signal
% computes 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',signal,freq,damp);`
computes the response spectrum associated to the time signal given in `signal`. Time derivatives can be obtained with option `-v` or `-a`. Time integration with option `+v` or `+a`. Pseudo derivatives with option `PseudoA` or `PseudoV`. `freq` and `damp` are frequencies (in Hz) and damping ratios vectors of interest for the response spectra. For example

```
wd=fileparts(which('d_ubeam'));
% read the acceleration time signal
bagnol_ns=fe_curve(['read' fullfile(wd,'bagnol_ns.cyt')]);

% read reference spectrum
bagnol_ns_rspec_pa= fe_curve(['read' fullfile(wd,'bagnol_ns_rspec_pa.cyt')]);
% compute response spectrum with reference spectrum frequencies
% vector and 5% damping
RespSpec=fe_curve('ResSpectrum PseudoA',...
                  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');
```

returny

If curve has a .Interp field, this interpolation is taken in account. If .Interp field is not present or empty, it uses a degree 2 interpolation by default.

To force a specific interpolation (over passing .interp field, one may insert the -linear, -log or -stair option string in the command.

To extract a curve curve_name and return the values Y corresponding to the input X, the syntax is

```
y = fe_curve('returny',model,curve_name,X);
```

Given a curve data structure, to return the values Y corresponding to the input X, the syntax is

```
y = fe_curve('returny',curve,X);
```

set

This command sets a curve in the model. 3 types of input are allowed:

- A data structure, model=fe_curve(model,'set',curve_name,data_structure)
- A string to interprete, model=fe_curve(model,'set',curve_name,string)
- A name referring to an existing curve (for load case only), model=fe_curve(model, 'set LoadCurve',load_case,chanel,curve_name). **This last behavior is obsolete** and should be replaced in your code by a more general call to fe_case SetCurve.

When you want to associate a curve to a load for time integration it is preferable to use a formal definition of the time dependence (if not curve can be interpolated or extrapolated).

The following example illustrates the different calls.

```
% Sample curve assignment to modal loads in a model
model=fe_time('demo bar'); q0=[];

% curve defined by a by-hand data structure:
c1=struct('ID',1,'X',linspace(0,1e-3,100), ...
     'Y',linspace(0,1e-3,100),'data',[],...
     'xunit',[],'yunit',[],'unit',[],'name','curve 1');
model=fe_curve(model,'set','curve 1',c1);
% curve defined by a string to evaluate (generally test fcn):
```

```
model=fe_curve(model,'set','step 1','TestStep t1=1e-3');
% curve defined by a reference curve:
c2=fe_curve('test -ID 100 ricker dt=1e-3 A=1');
model=fe_curve(model,'set','ricker 1',c2);
c3=fe_curve('test eval sin(2*pi*1000*t)'); % 1000 Hz sinus
model=fe_curve(model,'set','sin 1',c3);

% define Load with curve definition
LoadCase=struct('DOF',[1.01;2.01],'def',1e6*eye(2),...
            'curve',{{fe_curve('test ricker dt=2e-3 A=1'),...
                      'ricker 1'}});
model = fe_case(model,'DOFLoad','Point load 1',LoadCase);

% modify a curve in the load case
model=fe_case(model,'SetCurve','Point load 1','TestStep t1=1e-3',2);

% the obsolete but supported call was
model=fe_curve(model,'set LoadCurve','Point load 1',2,'TestStep t1=1e-3');

% one would prefer providing a name to the curve,
% that will be stacked in the model
model=fe_case(model,'SetCurve','Point load 1',...
 'my\_load','TestStep t1=1e-3',2);
```

Test ...

The `test` command handles a large array of analytic and tabular curves. In OpenFEM all parameters of each curve must be given in the proper order. In SDT you can specify only the ones that are not the default using their name.

When the abscissa vector (time, frequency, ...) is given as shown in the example, a tabular result is returned.

Without output argument the curve is simply plotted.

```
% Standard generation of parametered curves
fe_curve('test')  % lists curently implemented curves

t=linspace(0,3,1024); % Define abscissa vector
% OpenFEM format with all parameters (should be avoid):
C1=fe_curve('test ramp 0.6 2.5 2.3',t);
```

```
C2=fe_curve('TestRicker 2 2',t);

% SDT format non default parameters given with their name
%  definition is implicit and will be applied to time vector
%  during the time integration:
C3=fe_curve('Test CosHan f0=5 n0=3 A=3');
C4=fe_curve('testEval 3*cos(2*pi*5*t)');

% Now display result on time vector t:
C3=fe_curve(C3,t);C4=fe_curve(C4,t)
figure(1);plot(t,[C1.Y C2.Y C4.Y C3.Y]);
legend(C1.name,C2.name,C4.name,C3.name)
```

A partial list of accepted test curves follows

- `Testsin`, `Testcos`, `TestTan`, `TestExp`, accept parameters `T` period and `A` amplitude. `-stoptime Tf` will truncate the signal.
- `TestRamp t0=`*`t0`* `t1=`*`t1`* `Yf=`*`Yf`* has a ramp starting at zero until *t0* and going up to *Yf* at *t1*. The number of intermediate value can be controlled with the abscissa vector.
  To define a gradual load, for non linear static for example, a specific call with a `Nstep` parameter can be performed : `TestRamp NStep=`*`NStep`* `Yf=`*`Yf`*. For example, to define a 20 gradual steps to 1e-6 :`R1=fe_curve('TestRamp NStep=20 Yf=1e-6');`
- `TestRicker dt=`*`dt`* `A=`*`A`* `t0=`*`t0`* generates a Ricker function typically used to represent impacts of duration *dt* and amplitude *A*, starting from time *t0*.
- `TestSweep fmin=`*`fmin`* `fmax=`*`fmax`* `t0=`*`t0`* `t1=`*`t1`* generates a sweep cosine from *t0* to *t1*, with linear frequency sweeping from *f0* to *f1*.
  $Y = cos(2 * pi * \left(fmin + (fmax - fmin) * \frac{t-t0}{t1-t0}) * (t - t0)\right)$ for $t0 < t < t1$, $Y = 0$ elsewhere.
- `TestStep t1=`*`t1`* generates a step which value is one from time 0 to time *t1*.
- `TestNoise -window"`*`window`*`"` computes a time signal corresponding to a white noise, with the power spectrum density specified as the *window* parameter. For example `TestNoise "Box A=1 min=0 max=200"` defines a unitary power spectrum density from 0 Hz to 200 Hz.
- `TestBox A=`*`A`* `min=`*`min`* `max=`*`max`* generates a sample box signal from *min* to *max* abscissa, with an amplitude `A`.
- `TestEval` *str* generates the signal obtained by evaluating the string *str* function of `t`. For example `R1=fe_curve('Test eval sin(2*pi*1000*t)',linspace(0,0.005,501)); iiplot(R1)`

One can use fe_curve('TestList') to obtain a cell array of the test keywords recognized.

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. See sdtweb fe_curve('TestFrame') to open the function at this example.

timefreq

out=fe_curve('TimeFreq',Input,xf);
computes response of a system with given transfer functions FRF to time input Input. Sampling frequency and length of time signal Input must be coherent with frequency step and length of given transfer FRF.

```
% Plot time frequency diagrams of signals
fs=1024; sample_length=2;                    % 2 sec. long white noise
noise=fe_curve('noise',fs*sample_length,fs);% 1024 Hz of sampling freq.
[t,f,N]=fe_curve('getXTime',noise);

% FRF with resonant freq. 50 100 200 Hz, unit amplitude, 2% damping
xf=nor2xf(2*pi*[50 100 200].',.02,[1 ; 1 ; 1],[1 1 1],2*pi*f);

Resp=fe_curve('TimeFreq',noise,xf); % Response to noisy input
fe_curve('Plot',Resp); title('Time response');
```

Window ...

Use fe_curve window to list implemented windows. The general calling format is
win=fe_curve('Window Nb_pts Type Arg'); which computes a *Nb_pts* points window. The default is a symmetric window (last point at zero), the command option -per clips the last point of a $N+1$ long symmetric window.

For the exponential window the arguments are three doubles. 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)
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 [47].

Assemble [,-struct]

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');
[K,model,Case]=fe_case('assemble -matdes 2 1 NoT -cell',model);
SE=fe_cyclic('assemble -struct',model,[0 0 1000]); %
def=fe_eig({K{1:2},Case.T,model.DOF},[6 20 0]);% Non rotating modes
def2=fe_eig({K{1},SE.K{4},Case.T,model.DOF},[6 20 0]); % Rotating mode shapes
[def.data def2.data]
```

Note that the rotation speed can also be specified using a stack entry `model=stack_set(model, 'info', 'Omega',[0 0 1000])`.

Build ...

`model=fe_cyclic('build nsec epsl len',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.

The alternate command
`model=fe_cyclic('build nsec epsl len -intersect',model,'LeftNodeSelect')` is much faster but does not implement strict node tolerancing and may thus need an adjustement of `epsl` to higher values.

Command options are

- *nsec* is the optional number of sectors. An automatic determination of the number of angular sectors is implemented from the angle between the left and right interface nodes with the minimum radius. This guess may fail in some situtations so that the argument may be necessary.
- nsec=-1 is used for periodic structures and you should then provide the translation step. For periodic solutions,
  model=fe_cyclic('build -1 *tx ty tz* epsl *len* -intersect',model,'LeftNodeSelect')
  specifies 3 components for the spatial periodicity.
- Fix will adjust node positions to make the left and right nodes sets match exactly.
- epsl*len* gives the tolerance for edge node matching.
- -equal can be used to build a simple periodicity condition for use outside of fe_cyclic. This option is not relevant for cyclic symmetry.
- -ByMat is used to allow matching by MatId which allows for proper matching of coincident nodes.

```
model=demosdt('demo sector 5');
cf.model=fe_cyclic('build epsl 1e-6',model);
```

LoadCentrifugal

The command is used to build centrifugal loads based on an info,Omega stack entry in the form

```
data=struct('data',[0 0 1000],'unit','RPM');
model=stack_set(model,'info','Omega',data);
model=fe_cyclic('LoadCentrifugal',model);
```

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)
```

The basic functionality of this command is significantly extended in `fe_cyclicb ShaftEig` that is part of the SDT/Rotor toolbox.

Omega[,Group,GroupSet]

Handling of dynamic rotating bodies. **Warning** At the moment only one rotation vector can be defined. It can either be applied to the whole model or to specified groups. At low level, information is located in the `info,Omega` entry of an SDT model. This entry is a structure with fields

- `.data` provides the angular rotation vector whose norm is the angular velocity, defining the rotation axis.
- `.unit` provies the unit system associated to the amplitude, either`rad/s` or `RPM`.
- `.group` (optional) defines the model groups affected by the rotation, if omitted or left empty the whole model is affected.
- `.orig` (optional) defines the origin rotation (a point of the axis).

Command `Omega` provides the current data associated to a model.
`[omega,rot,data]=fe_cyclic('Omega',model);`
`model` is a standard SDT model. The outputs are `omega` the rotation vector, `rot` the rotation matrix, and `data` a reconstructed `info,Omega` stack entry based on the current state.

Commands `OmegaGroup` provides tools for definition of models with specific rotor areas.

- `OmegaGroupSet` provides an integrated definition forcing groups to be reset to conform with any `FindElt` selection. The specific group assignment is required due to low level assembly implementations.
  `model=fe_cyclic('OmegaGroupSet',model,list);`
  Input `model` is a standard SDT model, `list` is a three column cell-array with as many lines as declarations following the format `{FindEltStr, Amplitude, Axis, Orig;...}` respectively providing an element selection string, the angular velocity amplitude (scalar), the rotation axis (only the direction is used here), `nx,ny,nz`, and an origin point of the rotation axis `ox,oy,oz`. `data` can be directly placed as a stack entry named `info,OmegaData` in the model. The last

column can be omitted, in which case the origin considered is the global frame one. At the moment all rotation axes and amplitudes must be the same for all lines. The output `model` is then a model with separated groups for (one for each element type) affected to the rotation and with a new stack entry `info,Omega`. Command option `First` will force the new groups to be the first ones in the model.

- `OmegaGroup` is a lower level command without group modification.
  `model=fe_cyclic('OmegaGroup',model,sel,data);`
  Input `model` is a standard SDT model, `sel` is an element selection string, `data` is the omega structure with fields `.data` as defined at this command header.

See also

`fe_cyclicb`

fe_def

Purpose

Utilities for FEM related data structures.

Syntax

```
... = fe_def(def,'command', ... )
... = fe_def('command', ... )
```

Description

fe_defmainly provides utilities for SDT `def` structure handling. It is also used internally to perform parameter recovery.

Commands

CleanEntry

Returns the value of a parameter contained in an SDT button.

The entry can either be

- a structure of buttons in MATLAB format, for old GUI application, now called `DefBut`.
- a button in Java/SDT CinCell format for current GUI applications.
- a set of Java/SDT buttons EditT format for current GUI applications.

This call does not work for simple SDT buttons in MATLAB format (structures) due to the difficulty to distinguish between DefBut structures and button structures. To exploit this capability, one can place the button in a cell array, or a structure.

For treated button entries, the output is the current value of the button,

- if the button is a `pop`, a list restricted choice, that is characterized by its `type` field set to `pop` and the presence of a field `choices` and optionally `choicesTag`, the returned value is the currently selected `choicesTag` entry if existing, else the currently selected `choices`. In this case the button field `value` only is the index in the `choices`, `choicesTag` cell arrays. The output is mandatorily a string, in conformity with the content of `choices` and `choicesTag` that can only be horizontal cell array of strings.

- in other cases, the field `value` is returned. It is cast to the provided format given in the `format` field, either double with format `%g` or string with format `%s`.

For `EditT` entries, the output is a structure with as many fields as there are buttons in the EditT labelled with the names of the buttons. Each field contains each button value following the rules provided above.

For table entries (cell array with strings, CInCell...), the option `-CellField %s` gives back specific button fields like name or tooltip instead of current value. (see section 8.4.1 for the list of fields by button type)

```
% Define a SDT button that can be used in GUI
but=struct('type','string','format','%s','value','val1');
% no action on trivial button in MATLAB format
but=fe_def('cleanentry',but);
% use a cell array in this case
val=fe_def('cleanentry',{but}); val=val{1};

% Place the button in a DefBut
but1=struct('but1',but)
val=fe_def('cleanentry',but1);

% Transform button into Java format (CinCell)
but1=feval(cinguj('@toCinCell'),but);
% recover value with CleanEntry
val=fe_def('cleanentry',but1);

% Place Java button in an EditT object (a set of buttons)
r1j=cinguj('ObjEditJ',struct('but1',but1));
% recover the value of every button of the EditT at once in a struct
RO=fe_def('cleanentry',r1j);
```

DefEigOpt

`w=fe_def('DefEigOpt',mo1)` returns a `EigOpt` set of options for `fe_eig`for model `mo1`. If first searches for a field `info,EigOpt` in the model stack, or returns a default value, set to `[5 20 1e3]`.

DefFreq

`w=fe_def('DefFreq',DISK)` returns frequencies defined in the `info,Freq` entries using Hz units.

Exp

Performs modal expansion for def structures expressed on reduced DOF, if a reduction basis is provided.

- def=fe_def('Exp',TR,def) will restitute def on the non-reduced DOF of TR, a reduction basis expressed as a SDT def structure.
- def=fe_def('Exp',def), will assume that def contains the reduction basis in field def.TR to perform the expansion.

SubDef, SubDof, SubCh

def=fe_def('SubDef',def,ind); keeps deformations associated with ind, which a vector of indices or a logical vector (for example ind=def.data(:,1)<500 can be used to select frequencies below 500). Other fields of the def structure are truncated consistently.

def=fe_def('SubDof',def,DOF) is extracts a subset of DOFs based on defined DOF or with def=fe_def('subdofind',def,ind) indices (again either values or logicals). You can also specify DOFs to be removed with def=fe_def('SubDofRem',def,DofRemoved).

This command is partially redundant with feutilb PlaceInDof called with
def2 = feutilb('PlaceInDof',DOF,def). The main difference is the ability to add zeros (use DOF larger than def.DOF) and support sens structures.

fe_def('SubDofInd-Cell',def,ind_dof,ind_def) returns a clean cell array listing selected DOFs and responses. This is typically used to generate clean tables.

fe_def('SubChCurve',def,{'lab',index}) is similar to SubDof but allows but supports more advanced selection for multi-dimensional curves. This command is not fully documented.

```
C1=demosdt('Curve curved5'); % Sample 5D curve
C2=fe_def('subChCurve',C1,{'Time',1:10;'RPM',1:2});
```

fe_eig

Purpose

Computation of normal modes associated to a second order undamped model.

Syntax

```
def = fe_eig(model,EigOpt)
def = fe_eig({m,k,mdof},EigOpt)
def = fe_eig({m,k,T,mdof},EigOpt)
[phi, wj] = fe_eig(m,k)
[phi, wj, kd] = fe_eig(m,k,EigOpt,imode)
```

Description

The normal modeshapes `phi`=ϕ and frequencies `wj=sqrt(diag(`Ω^2`))` are solution of the undamped eigenvalue problem (see section 5.2)

$$-[M]\{\phi_j\}\omega_j^2+[K]\{\phi_j\}=\{0\} \tag{10.13}$$

and verify the two orthogonality conditions

$$[\phi]^T[M]_{N\times N}[\phi]_{N\times N}=[I]_{N\times N} \text{ and } [\phi]^T[K][\phi]=\left[\backslash\Omega_j^2\backslash\right] \tag{10.14}$$

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 option vector `EigOpt` can be supplied explicitely or set using `model=stack_set(model, 'info','EigOpt',EigOpt)`. Its format is

`[method nm Shift Print Thres]` (default values are `[2 0 0 0 1e-5]`)

- method
  - **2** full matrix solution. Cannot be used for large models, used by default when the number of searched modes exceed 25% of the matrix size.
  - **5 default** Lanczos solver is an iterative solver with problem size scalability and higher robustness with convergence checks. To turn off convergence check add 2000 to the option (2105, 2005, ...), otherwise a maximum of 5 convergence iterations is performed. You can tune this value by setting the 9^{th} value of the opt vector to the desired number. opt=[5 100 1e3 1 0 0 0 0 1];.
  - **6** IRA/Sorensen solver. Faster than 5 but less robust, issues are known for multiple modes, very close frequencies, or when computing a large number of modes.
  - **50** Callback to let the user specify an external solver method using setpref('SDT','ExternalEig').
  - The other methods are left for reference but should not be used,
    - **105, 106, 104** same as 5, 6, 4 methods but no initial DOF renumbering. This is useless with the default ofact('methodspfmex') which renumbers at factorization time.
    - **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 (this is limited to 12 modes with method 5). The easiest way to handle fmax at the moment is to call fe_def SubDef after fe_eigThe sample syntax is then def=fe_eig(model,[5 50 1e3]); def=fe_def('subdef',def,find(def.data(:,1)<=fmax));. One thus has to estimate the relevant number of modes necessary beforehand.

- 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 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.
- `Method 6` calls `eigs` (ARPACK) properly and cleans up results. This solver sometimes fails to reach convergence, use `method 5` then.
- The subspace iteration and Lanczos algorithms are rather free interpretation of the standard algorithms (see Ref. [37] for example).
- 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. For large frequency bands, consider using a shift at 75% of the largest estimated frequency, using $-(0.75 * 2 * pi * f_max)^2$.
- DOFs with zero values on the stiffness diagonal are eliminated by default. You can bypass this behavior by giving a shift with unit imaginary value (`eigopt(3)=1e3+1i` for example).
- For performance, optimization matters, please refer to section section 4.8.7 .

Example

Here is an example containing a high level call

```
model =demosdt('demo gartfe');
cf=feplot;cf.model=model;
cf.def=fe_eig(model,[5 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},[6 20 1e3]);fecom chc10
```

See also

fe_ceig, fe_mk, nor2ss, nor2xf

fe_exp

Purpose

Expansion of experimental modeshapes.

Syntax

```
dExp = fe_exp('method',ID,Sens,FEM);
dExp = fe_exp('method',ID,SE);
```

Description

A unified perspective on interpolation and the more advanced finite element based expansion methods are discussed in the tutorial 3.3. 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.

`Subspace, Modal, Serep`

Subspace expansion solves a problem of the form

$$\{q_{exp}\} = [T]\{q_r\} \text{ with } \{q_r\} = argmin \|y_{test} - [cT]\{q_r\}\|^2 \tag{10.15}$$

Modal or SEREP expansion is a subspace based expansion using the subspace spanned by low frequency target modes (stored in `TR` in the `def` format). With a sensor configuration defined (`sens` defined using `fe_sens`), a typical call would be

```
[model,Sens,ID,FEM]=demosdt('demopairmac');  %sdtweb demosdt('demopairmac')
TR=fe_def('subdef',FEM,1:20); % Subspace containing 20 modes
dex1 = fe_exp('Subspace',ID,Sens,TR);
cf=feplot(model);
cf.def(1)=fe_def('subdef',FEM,7:20); % Rigid not in FEM
cf.def(2)=dex1; fecom('show2def');
```

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.

Another traditional approach to build subspaces is to generate the solutions by **mathematical interpolation**. `fe_sens WireExp` provides such a strategy. For a basic example of needed data structures, one considers the following case of a structure with 3 nodes. 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

```
TR=struct('DOF',[1.01;2.01;3.01], ... % DOFs where subspace is defined
  'def',[1 0;3/4 1/4;0 1]); % Each .def column associated with a vector
% sdtweb sens#sensstruct % manual definition of a sens structure
sens=struct('cta',[1 0 0;0 0 1],'DOF',[1.01;2.01;3.01])
% Sample test shapes
ID=struct('def',eye(2),'DOF',[1.01;3.01]);
dexp = fe_exp('Subspace',ID,sens,TR) % Expansion
```

For expansion of this form, T (stored in `TR.def`) 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.

`Subspace-Orth` can be used to impose that an orthogonal linear combination of the modes is used for the expansion. This is motivated for cases where both test and analysis modeshapes are mass normalized and will provide mass orthonormal expanded modeshapes [48]. In practice it is rare that test results are accurately mass normalized and the approach is only implemented for completeness.

Static

Static expansion is a subspace method where the subspace is associated with the static response to enforced motion or load at sensors. While you can use `fe_reduc Static` to build the subspace (or import a reduced subspace from an external code), a direct implementation for general definition of sensors is provided in `fe_exp`.

The main limitation with static expansion is the existence of a frequency limit (first frequency found when all sensors are fixed). These modes can be returned as a second argument to the `Static` command as illustrated below. If the first frequency is close to your test bandwidth, you should consider using dynamic expansion or possibly add sensors, see [49].

```
[model,Sens,ID,FEM]=demosdt('demopairmac');  %sdtweb demosdt('demopairmac')
[TR,dfix]=fe_exp('static',model,Sens);  % Build static subpace
dex1 = fe_exp('Subspace',ID,Sens,TR);
cf=feplot(model,dex1); % Expanded mode
cf=feplot(model,dfix); % Fixed interface mode
```

In the present case, the fixed sensor mode at 44 Hz indicates that above that frequency, additional sensors should be added in the y direction for proper static expansion.

When many sensors and model reduction are used as in the example below, Lagrange resolution should be preferred to elimination, using options `'Solve','lagrange'` and possibly adjusting the conditionning scalar `'pcond',1e-4`.

SE reduced superelements for expansion

All the expansion methods can be applied on reduced models.

```
% Further illustrations in gartco demo
[model,Sens,ID,FEM]=demosdt('demopairmac');  %sdtweb demosdt('demopairmac')
 RA=struct('wd',sdtdef('TempDir'), ...
   'Reset',0, ...           % 1 to use reset
   'oProp',{{}}, ...
   'EigOpt',[5 20 1e3], ... % Eigenvalue options
   'SensName','test', ...   % Sensor set for expansion
   'OutName','Gart_exp');   % Root of file name
% Generate or reload reduced model with modes & static
SE=fe_exp('mode+sens',model,RA);

RA=struct('Solve','lagrange','pcond',1e-4); % Solve using lagrange multipliers
[dex1,dfix] = fe_exp('Static',ID,SE,RA); % Static
dex3 = fe_exp('dynamic',ID,SE,RA); % Dynamic on reduced model
cf=feplot(model);
RO=struct('type','mdrewe','gamma',1e6, ...
  'cf',cf,'view',{{'fe_exp','viewEnerKDens',cf,'out1'}}); % feplot for display
[dex4,err] = fe_exp('mdre',ID,SE,RO); % MDRE-WE
```

Dynamic

Dynamic expansion is supported at the frequency of each deformation to be expanded using either full or reduced computations.

```
% Further illustrations in gartco demo
[model,Sens,ID,FEM]=demosdt('demopairmac');  %sdtweb demosdt('demopairmac')
dex1 = fe_exp('Dynamic',ID,Sens,model); % Dynamic full model
```

The preferred strategy is to build a reduced model `SE` containing normal and attachment modes. When many sensors are used Lagrange resolution should be preferred to elimination as shown in the example above.

MDRE, MDRE-WE

Minimum dynamic residual expansion (MDRE) is currently only implemented for normal mode-shape expansion. Furthermore, computational times are generally only acceptable for the reduced

basis form of the algorithm as illustrated above. Note that the result may incorrect due to poor conditionning with a large number of sensors.

MDRE-WE (Minimum dynamic residual expansion with measurement error) is adjusted by the relative weighting γ_j between model and test error in (3.9)

$$min_{q_{j,ex}} \|R(q_{j,ex})\|_K^2 + \gamma_j \epsilon_j \tag{10.16}$$

Fields of the option structure are

- `.type='mdrewe'`
- `.gamma` weighting coefficient.
- `.cf` feplot figure for display.
- `.view` callback executed for energy display.

The first output argument is the expanded modeshape, the second the displacement residual which shown high energy concentration in locations where the model is wrong or the test very far from the model (which can occur when the test is wrong/noisy).

See also

`fe_sens`, `fe_reduc`, section 3.3 , `gartco` demo.

fe_gmsh

Purpose

GMSH interface. You can download GMSH at `http://www.geuz.org/gmsh/` and tell where to find GMSH using

Syntax

```
setpref('OpenFEM','gmsh','/path_to_binary/gmsh.exe') % Config
model=fe_gmsh(command,model,...)
model=fe_gmsh('write -run','FileName.stl')
```

Description

The main operation is the automatic meshing of surfaces.

Example

This example illustrates the automatic meshing of a plate

```
FEnode = [1 0 0 0  0 0 0; 2 0 0 0  1 0 0; 3 0 0 0  0 2 0];
femesh('objectholeinplate 1 2 3 .5 .5 3 4 4');
model=femesh('model0');
model.Elt=feutil('selelt seledge ',model);
model.Node=feutil('getnode groupall',model);
model=fe_gmsh('addline',model,'groupall');
model.Node(:,4)=0; % reset default length
mo1=fe_gmsh('write del.geo -lc .3 -run -2 -v 0',model);
feplot(mo1)
```

This other example makes a circular hole in a plate

```
% Hole in plate :
model=feutil('Objectquad 1 1',[0 0 0; 1 0 0;1 1 0;0 1 0],1,1); %
model=fe_gmsh('addline -loop1',model,[1 2; 2 4]);
model=fe_gmsh('addline -loop1',model,[4 3; 3 1]);
model=fe_gmsh('AddFullCircle -loop2',model,[.5 .5 0; .4 .5 0; 0 0 1]);

model.Stack{end}.PlaneSurface=[1 2];
mo1=fe_gmsh('write del.geo -lc .02 -run -2 -v 0',model)
feplot(mo1)
```

To allow automated running of GMSH from MATLAB, this function uses a `info,GMSH` stack entry with the following fields

`.Line`	one line per row referencing `NodeId`. Can be defined using `addline` commands.
`.Circle`	define properties of circles.
`.LineLoop`	rows define a closed line as combination of elementary lines. Values are row indices in the `.Line` field. One can also define `LineLoop` from circle arcs (or mixed arcs and lines) using a cell array whose each row describes a lineloop as {*'LineType'*,LineInd,...} where *LineType* can be `Circle` or `Line` and `LineInd` row indices in corresponding `.Line` or `.Circle` field.
`.TransfiniteLines`	Defines lines which seeding is controlled.
`.PlaneSurface`	rows define surfaces as a combination of line loops, values are row indices in the `.LineLoop` field. Negative values are used to reverse the line orientation. 1st column describes the exterior contour, and followings the interiors to be removed. As `.PlaneSurface` is a matrix, extra columns can be filled by zeros.
`.EmbeddedLines`	define line indices which do not define mesh contours but add additional constrains to the final mesh (see Line In Surface in the `gmsh` documentation.
`.SurfaceLoop`	rows define a closed surface as combination of elementary surfaces. Values are row indices in the `.PlaneSurface` field.

The local mesh size is defined at nodes by GMSH. This is stored in column 4 of the `model.Node`. Command option `-lc`*`val`* in the command resets the value *`val`* for all nodes that do not have a prior value.

`Add...`

Typical calls are of the form `[mdl,RO]=fe_gmsh('Add Cmd',mdl,data)`. The optional second output argument can be used to obtain additional information like the `LoopInfo`. Accepted command options are

- `-loop i` is used to add the given geometries and append the associated indices into the `LineLoop(i)`.
- `FullCircle` defines a circle defined using `data` with rows giving center coordinates, an edge node coordinates and the normal in the last row. 4 arcs of circle are added. In the `LineLoop` field the entry has the form `{'Circle',[ind1 ind2 ind3 ind4]}` where `indi` are the row indices of the 4 arcs of circle created in `.Circle` field.
- `CircleArc` defines a circle arc using `data`

  - 3x3 matrix, with 1rst row giving center coordinates, second and third rows are respectively the first and second edges defined by node coordinates.
  - 3x1 vector, giving the 3 NodeId (center, 1st and 2nd edge) as a column instead of x y z.
  - with a `-tangent1` option, 3x3 matrix whose 1st row defines a tangent vector of the circle arc at the 1st edge node (resp. at the second edge node with the option `-tangent2`). 2nd row defines the 1st edge node coordinates and third row the 2nd edge node coordinate.

- `Disk` ...

- `Line` accepts multiple formats. `data` can be a 2 column matrix which each row defines a couple of points from their `NodeId`.

  `data` can also be a 2 by 3 matrix defining the coordinates of the 2 extremities.

  `data` can also be a string defining a line selection.

  - It is possible to specify a seeding on the line for further meshing operation using additional arguments `seed` and the number of nodes to seed on the line.
    E.g.: `mdl=fe_gmsh('AddLine',mdl,data,'seed',5);` will ask `gmsh` to place 5 nodes on each line declared in `data`.
  - It is possible to define line constrains in mesh interiors using embedded lines (depending on the `gmsh` version). `mdl=fe_gmsh('AddLine',mdl,data,'embed',1);` will thus declare the edges found in data not as line loops defining surfaces, but as interior mesh constrains. This feature is only supported for lines specified as selections.

- `AddLine3` can be used to declare splines instead of lines in the geometry. For this command to work, `beam3` elements must be used, so that a middle node exists to be declared as the spline control point. For this command, data can only be an element string selection.

config

The `fe_gmsh` function uses the `OpenFEM` preference to launch the GMSH mesher.

```
setpref('OpenFEM','gmsh','$HOME_GMSH/gmsh.exe')
```

Ver

Command `Ver` returns the version of `gmsh`, the version is transformed into a double to simplify hierarchy handling (*e.g.* version 2.5.1 is transformed into 251). This command also provides a good test to check your `gmsh` setup as the output will be empty if `gmsh` could not be found.

Read

`fe_gmsh('read FileName.msh')` reads a mesh from the GMSH output format.

Write

`fe_gmsh('write FileName.geo',model);` writes a model (`.Node`, `.Elt`) and geometry data in `model.Stack'info','GMSH'` into a `.geo` file which root name is specified as `FileName` (if you use `del.geo` the file is deleted on exit).

- Command option `-lc` allows specifying a characteristic length. You can also define a nodewise characteristic length by setting non zero values in `model.Node(:,4)`.
- Command option `-multiple` can be used for automated meshing of several closed contours. The default behavior will define a single Plane Surface combining all contours, while `-multiple` variant will declare each contour as a single Plane Surface.
- Command option `-keepContour` can be used to force `gmsh` not to add nodes in declared line objects (`Transfinite Line` feature).
- Command option `-spline` can be used (when lines have been declared using command `AddLine3` from `beam3` elements) to write spline objects instead of line objects in the `.geo` file
- `.stl` writing format is also supported, by using extension `.stl` instead of `.geo` in the command line.
- Command option `-run` allows to run `gmsh` on the written file for meshing. All characters in the command after `-run` will be passed to the `gmsh` batch call performed. `fe_gmsh` then outputs the model processed by `gmsh`, which is usually written in `.msh` file format.

  All text after the `-run` is propagated to GMSH, see sample options below.
  It also possible to add a different output file name *NewFile.msh*, using `model=fe_gmsh('write NewFile.msh -run','FileName.stl')`.
- Conversion of files through `fe_gmsh` into `.msh`, or SDT/OpenFEM format is possible, for all input files readable by `gmsh`. Use command option `-run` and specify in second argument the file name.
  For example: `model=fe_gmsh('write -run','FileName.stl')` convert `.stl` to `.mesh` then open into SDT/OpenFem. Some warning can occur if no `FileName.mesh` is given, but without effect on the result.

Known options for the run are

- -1 or -2 or -3) specifies the meshing dimension.
- -order 2 uses quadratic elements.
- -v 0 makes a silent run.
- -clmax float sets maximum mesh size, -clmin float for minimum.

From a geometry file the simplest meshing call is illustrated below

```
filename=demosdt('download-back http://www.sdtools.com/contrib/component8.step')
RO=struct( ... % Predefine materials
  'pl',m_elastic('dbval -unit TM 1 steel'), ...
  'sel','selelt eltname tetra10', ... % Elements to retain at end
  'Run','-3 -order 2 -clmax 3 -clmin 2 -v 0');  %RunCommand
 model=fe_gmsh('write',filename,RO);
```

It is also possible to write GMSH post-processing command lines, written at the end of the file (see the GMSH documentation) by providing a cell array (one cell by command line) in the field .Post of the RO structure.

See also

missread

fe_load

Purpose

Interface for the assembly of distributed and multiple load patterns

Syntax

```
Load = fe_load(model)
Load = fe_load(model,Case)
Load = fe_load(model,'NoT')
Load = fe_load(model,Case,'NoT')
```

Description

fe_load is used to assemble loads (left hand side vectors to FEM problems). Loads are associated with case structures with at least a Case.Stack field giving all the case entries. Addition of entries to the cases, it typically done using fe_case.

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). In general simultaneous assembly of matrices and loads detailed in section 4.8.8 is preferable.

The option 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.

curve

The frequency or time dependence of a load can be specified as a data.curve field in the load case entry. This field is a cell array specifying the dependence for each column of the applied loads.

Each entry can be a curve data structure, or a string referring to an existing curve (stored in the model.Stack), to describe frequency or time dependence of loads.

Units for the load are defined through the .lab field (in $\{F\} = [B]\{u\}$ one assumes u to be unitless thus F and B have the same unit systems).

DofLoad, DofSet

Loads at DOFs DofLoad *and prescribed displacements* DofSet entries are described by the following data structure

data.DOF	column vector containing a DOF selection
data.def	matrix of load/set for each DOF (each column is a load/set case and the rows are indexed by Case.DOF). With two DOFs, def=[1;1] is a single input at two DOFs, while def=eye(2) corresponds to two inputs.
data.name	optional name of the case
data.lab	optional cell array giving label, unit label , and unit info (see fe_curve DataType) for each load (column of data.def)
data.curve	see fe_load curve

Typical initialization is illustrated below

```
% Applying a load case in a model
 model = femesh('testubeam plot');
 % Simplified format to declare unit inputs
 model=fe_case(model,'DofLoad','ShortTwoInputs',[362.01;258.02]);

 % General format with amplitudes at multiple DOF
 % At node 365, 1 N in x and 1.1 N in z
 data=struct('DOF',[365.01;365.03],'def',[1;1.1]);
 data.lab=fe_curve('datatype',13);
 model=fe_case(model,'DofLoad','PointLoad',data);

 Load = fe_load(model);
 feplot(model,Load); fecom(';scaleone;undefline;ch1 2') % display
```

When sensors are defined in SDT, loads collocated with sensors can be defined using sensor DofLoadSensDof.

FVol

FVol entries use data is a structure with fields

data.sel	an element selection (or amodel description matrix but this is not acceptable for non-linear applications).
data.dir	a 3 by 1 cell array specifying the value in each global direction x, y, z. Alternatives for this specification are detailed below . The field can also be specified using .def and .DOF fields.
data.lab	cell array giving label, unit label , and unit info (see fe_curve DataType) for each load (column of data.def)
data.curve	see fe_load curve

Each cell of Case.dir can give a constant value, 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 and you can use the coordinates x,y,z for your evaluation. The transformation to a vector defined at model.DOF is done using vect=elem0('VectFromDir',model,r1,model.DOF), you can look the source code for more details.

For example

```
% Applying a volumic load in a model
 model = femesh('testubeam');
 data=struct('sel','groupall','dir',[0 32 0]);
 data2=struct('sel','groupall','dir',{{0,0,'(z-1).^3.*x'}});
 model=fe_case(model,'FVol','Constant',data, ...
                     'FVol','Variable',data2);
 Load = fe_load(model);
 feplot(model,Load);fecom(';colordataz;ch2'); % display
```

Volume loads are implemented for all elements, you can always get an example using the elements self tests, for example [model,Load]=beam1('testload').

Gravity loads are not explicitly implemented (care must be taken considering masses in this case and not volume). You should use the product of the mass matrix with the rigid body mode corresponding to a uniform acceleration.

FSurf

FSurf entries use data a structure with fields

data.sel	a vector of NodeId in which the faces are contained (all the nodes in a loaded face/edge must be contained in the list). data.sel can also contain any valid node selection (using string or cell array format). the optional data.eltsel field can be used for an optional element selection to be performed before selection of faces with feutil('selelt innode',model,data.sel). The surface is obtained using

```
% Surface selection mechanism performed for a FSurf input
 if isfield(data,'eltsel');
  mo1.Elt=feutil('selelt',mo1,data.eltsel);
 end
 elt=feutil('seleltinnode',mo1, ...
     feutil('findnode',mo1,r1.sel));
```

data.set	Alternative specification of the loaded face by specifying a face set name to be found in model.Stack
data.def	a vector with as many rows as data.DOF specifying a value for each DOF.
data.DOF	DOF definition vector specifying what DOFs are loaded. Note that pressure is DOF .19 and generates a load opposite to the outgoing surface normal. Uniform pressure can be defined using wild cards as show in the example below.
data.lab	cell array giving label, unit label ,and unit info (see fe_curve DataType) for each load (column of data.def)
data.curve	see fe_load curve
data.type	string giving 'surface' (default) or 'edge' (used in the case of 2D models where external surfaces are edges)

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

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

Or an alternative call with the cell array format for data.sel

```
% Applying a surfacing load case in a model using node lists
 data=struct('eltsel','withnode {z>1.25}','def',1,'DOF',.19);
 NodeList=feutil('findnode x==-.5',model);
 data.sel={'','NodeId','==',NodeList};
 model=fe_case(model,'Fsurf','Surface load',data);
 Load = fe_load(model); feplot(model,Load);
```

Alternatively, one can specify the surface by referring to a set entry in model.Stack, as shown in the following example

```
% Applying a surfacing load case in a model using sets
 model = femesh('testubeam plot');

 % Define a face set
 [eltid,model.Elt]=feutil('eltidfix',model);
 i1=feutil('findelt withnode {x==-.5 & y<0}',model);i1=eltid(i1);
 i1(:,2)=2; % fourth face is loaded
 data=struct('ID',1,'data',i1);
 model=stack_set(model,'set','Face 1',data);

 % define a load on face 1
 data=struct('set','Face 1','def',1,'DOF',.19);
 model=fe_case(model,'Fsurf','Surface load',data);
 Load = fe_load(model); feplot(model,Load)
```

The current trend of development is to consider surface loads as surface elements and transform the case entry to a volume load on a surface.

See also

fe_c, fe_case, fe_mk

fe_mat

Purpose

Material / element property handling utilities.

Syntax

```
out = fe_mat('convert si ba',pl);
typ=fe_mat('m_function',UnitCode,SubType)
[m_function',UnitCode,SubType]=fe_mat('type',typ)
out = fe_mat('unit')
out = fe_mat('unitlabel',UnitSystemCode)
[o1,o2,o3]=fe_mat(ElemP,ID,pl,il)
```

Description

Material definitions can be handled graphically using the Material tab in the model editor (see section 4.5.1). 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

The convert command supports conversions from unit1 to unit2 with the general syntax pl_converted = fe_mat('convert unit1 unit2',pl);.

For example convert from SI to BA and back

```
% Sample unit convertion calls
 mat = m_elastic('default'); % Default is in SI
 % convert mat.pl from SI unit to BA unit
 pl=fe_mat('convert SIBA',mat.pl)
 % for section properties IL, you need to specify -il
 fe_mat('convert -il MM',p_beam('dbval 1 circle .01'))
 % For every system but US you don't need to specify the from
 pl=fe_mat('convert BA',mat.pl)
 % check that conversion is OK
 pl2=fe_mat('convert BASI',pl);
 fprintf('Conversion roundoff error : %g\n',norm(mat.pl-pl2(1:6))/norm(pl))
 fe_mat('convertSIMM') % Lists defined units and coefficients
 coef=fe_mat('convertSIMM',2.012) % conversion coefficient for force/m^2
```

Supported units are either those listed with `fe_mat('convertSIMM')` which shows the index of each unit in the first column or ratios of any of these units. Thus, 2.012 means the unit 2 (force) divided by unit 12 (surface), which in this case is equivalent to unit 1 pressure.

`out=fe_mat('unitsystem')` returns a `struct` containing the information characterizing standardized unit systems supported in the universal file format.

ID		Length and Force	ID		
1	SI	Meter, Newton	7	IN	Inch, Pound force
2	BG	Foot, Pound f	8	GM	Millimeter, kilogram force
3	MG	Meter, kilogram f	9	TM	Millimeter, Newton
4	BA	Foot, poundal	9	US	User defined
5	MM	Millimeter, milli-newton			
6	CM	Centimeter, centi-newton			

Unit codes 1-8 are defined in the universal file format specification and thus coded in the material/element property type (column 2). Other unit systems are considered user types and are associated with unit code 9. With a unit code 9, `fe_mat convert` commands must give both the initial and final unit systems.

`out=fe_mat('unitlabel',UnitSystemCode)` returns a standardized list of unit labels corresponding in the unit system selected by the `UnitSystemCode` shown in the table above.

When defining your own properties material properties, automated unit conversion is implemented automatically through tables defined in the `p_fun PropertyUnitType` command.

GetPl GetIl

`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`.

Get[Mat,Pro]

`r1 = fe_mat('GetMat` *Param*`',model)` This command can be used to extract given parameter *Param* value in the model properties. For example one can retrieve density of matid 111 as following `rho=fe_mat('GetMat 111 rho',model);`

Set[Mat,Pro]

```
r1 = fe_mat('SetMat MatId Param=value',model)
r1 = fe_mat('SetPro ProId Param=value',model)
```

This command can be used to set given parameter *Param* at the value *value* in the model properties. For example one can set density of matid 111 at 5000 as following
`rho=fe_mat('SetMat 111 rho=5000',model);`

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 developers only and you should look at the source code of each element.

See also

`m_elastic`, `p_shell`, element functions in chapter 9

fe_mknl, fe_mk

Purpose

Assembly of finite element model matrices.

Syntax

```
[m,k,mdof] = fe_mknl(model);
[Case,model.DOF]=fe_mknl('init',model);
mat=fe_mknl('assemble',model,Case,def,MatType);
```

Description

The exact procedure used for assembly often needs to be optimized in detail to avoid repetition of unnecessary steps. SDT typically calls an internal procedure implemented in `fe_caseg Assemble` and detailed in section 4.8.8 . This documentation is meant for low level calls.

`fe_mknl` (and the obsolete `fe_mk`) take models and return assembled matrices and/or right hand side vectors.

Input arguments are

- `model` a model data structure describing nodes, elements, material properties, element properties, and possibly a `case`.
- `case` data structure describing loads, boundary conditions, etc. This may be stored in the model and be retrieved automatically using `fe_case(model,'GetCase')`.
- `def` a data structure describing the current state of the model for model/residual assembly using `fe_mknl`. `def` is expected to use model DOFs. If `Case` DOFs are used, they are reexpanded to model DOFs using `def=struct('def',Case.T*def.def,'DOF',model.DOF)`. This is currently used for geometrically non-linear matrices.
- `MatType` or `Opt` describing the desired output, appropriate handling of linear constraints, etc.

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

```
% Low level assembly call with or without constraint resolution
 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 7.14 .

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 7.14 , resolution calls fe_case GetT, Case.GroupInfo fields (detailed below) and Case.MatGraph (preallocated sparse matrix associated with the model topology for optimized (re)assembly). Case.GroupInfo is a cell array with rows giving information about each element group in the model (see section 7.15.3 for details).

Command options are the following

- NoCon 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).

- Keep can be used to prevent changing the model.DOF DOF list. This is typically used for submodel assembly.
- -NodePos saves the NodePos node position index matrix for a given group in its EltConst entry.
- -gstate is used force initialization of group stress entries.
- new will force a reset of Case.T.

The initialization phase is decomposed into the following steps

1. Generation of a complete list of DOFs using the feutil('getdof',model) call.
2. get the material and element property tables in a robust manner (since some data can be replicated between the pl,il fields and the mat,pro stack entries. Generate node positions in a global reference frame.
3. For each element group, build the GroupInfo data (DOF positions).
4. 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 relevant information in the integ and constit tables.

   This is typically done using an [integ,constit,ElMap]=ElemF('integinfo') command, which in most cases is really being passed directly to a p_fun('BuildConstit') command.

   ElMap can be a structure with fields beginning by RunOpt_, Case_ and eval which allows execution of specific callbacks at this stage.
5. For each element group, perform other initializations as defined by evaluating the callback string obtained using elem('GroupInit'). For example, initialize integration rule data structures EltConst, define local bases or normal maps in InfoAtNode, allocate memory for internal state variables in gstate, ...
6. 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

```
% Low level assembly calls
 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 identifiers

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 (assumes a static state given in the call. In SDT calls (see section 4.8.8), the case entry 'curve','StaticState' is used to store the static state.
- 3 viscous damping. Uses info,Rayleigh case entries if defined, see example in section 5.3.2 .
- 4 hysteretic damping. Weighs the stiffness matrices associated with each material with the associated loss factors. These are identified by the key word Eta in PropertyUnitType commands.
- 7 gyroscopic coupling in the body fixed frame, 70 gyroscopic coupling in the global frame. 8 centrifugal softening.
- 9 is reserved for non-symmetric stiffness coupling (fluid structure, contact/friction, ...);

- 20 to assemble a lumped mass instead of a consistent mass although using common integration rules at Gauss points.
- 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. For pre-stressed beams, 300 modifies InfoAtNode=Case.GroupInfo{jGroup,7}.
- -1, -1.1 submodel selected by parameter, see section 4.8.8 .
- -2, -2.1 specific assembly of superelements with label split, see section 4.8.8 .

NodePos

NodePos=fe_mknl('NodePos',NNode,elt,cEGI,ElemF) is used to build the node position index matrix for a given group. ElemF can be omitted. NNode can be replaced by node.

nd

nd=fe_mknl('nd',DOF); is used to build and optimized object to get indices of DOF in a DOF list.

OrientMap

This command is used to build the InfoAtNode entry. The 'Info','EltOrient' field is a possible stack entry containing appropriate information before step 5 of the init command. The preferred mechanism is to define an material map associated to an element property as illustrated in section 7.13 .

of_mk

of_mk is the mex file supporting assembly operations. You can set the number of threads used with of_mk('setomppro',8).

obsolete

Syntax

```
model      = fe_mk(model,'Options',Opt)
[m,k,mdof] = fe_mk( ... ,[0       OtherOptions])
[mat,mdof] = fe_mk( ... ,[MatType OtherOptions])
```

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 sdtdef tempname command. 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.

[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 9, 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 orthonormal condition

$$[To]^T [M]_{N\times N} [To]_{N\times NM} = [I]_{NM\times NM} \tag{10.17}$$

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 $\left[\backslash\Omega_j^2\backslash\right]$ `=diag(wr.^2)`)

$$[\phi]^T [K] [\phi] = \left[\backslash\Omega_j^2\backslash\right]_{NM\times NM} \tag{10.18}$$

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_quality

Purpose

Mesh quality measurement tools

Description

This function provides mesh quality measurement, visualization and report tools. Badly shaped elements are known to cause computation error and imprecision, and basic geometric tests can help to acknowledge such property. Every element cannot be tested the same way therefore the `lab` command presents the tests available for each kind. The geometric measurements performed are described in the following section.

An integrated call is provided for `feplot`,

```
fe_quality(cf.mdl);
```

This call performs all test available and opens a GUI allowing the user to customize the views.

Available tests

Degenerate

Degenerated elements have overlaying nodes which is generally unwanted. The set is automatically generated when such elements are detected.

Jacobian

This test computes the minimum Jacobian for each element and detects negative values. It is directly related to the element volume so that a wrapped element would show such pattern. The set is generated only if elements with negative Jacobian are detected.

AspectRatio

This test can be applied to any kind of element. It computes the ratio of the longest edge length to the shortest one. Thus a good element will have an aspect ratio close to one while a badly shaped element will have a greater aspect ratio. The Default tolerance for visualization is set to 2.

MaxIntAng

This test can be applied to triangle and quadrangle elements (tria3, tria6, quad4, quad8, quadb). It measures the greatest angle in an element which is an indication of element distortion. The default tolerance is set to 110 degrees.

GammaK

This test is applied to triangle elements (tria3, tria6). It computes the ratio between the radius of the inscribed circle and the circumcircle. This indicator is named γ_K and is bounded between 0 and 1. Well shaped elements will have a γ_K coefficient close to one. Degenerated triangles show $\gamma_K = 0$. The default tolerance is set to 0.5.

MidNodeEgde

This test is applied to quadratic triangles (tria6). It measures the distance of the middle nodes to the edge nodes. The ratio between the distance from the middle node to the first edge node (l_1) and the distance from the middle node to the second edge node (l_2) is computed for each element as $MNE = max_{i=1...3}(\frac{max(l_{1i}/l_{2i}, l_{2i}/l_{1i})}{min(l_{1i}/l_{2i}, l_{2i}/l_{1i})})$ The default tolerance is set to 1.5.

MaxAngleMid2Edge

This test is applied to quadratic triangles (tria6). It measures the distortion of the edges by computing the maximum angle between the straight edge (between both edge extreme nodes) and the actual edges through the middle node. The maximum over the whole triangle is output, the default tolerance is set to 30 degrees.

Taper

This test is applied to 2D quadrangle elements (quadb). It compares the areas of the 4 triangles formed by the diagonals and each edge to the area of the full quadrangle. The exact computation is $max(\frac{2\mathcal{A}_i}{\mathcal{A}_K})$. Thus a well shaped element will show a taper ratio close to 0.5, while a badly shaped element can have taper ratios over 1. The default tolerance is set to 0.8.

Skew

This test is applied to quadrangle elements (quad4, quad8, quadb). It evaluates the element distortion by measuring the angle formed by the diagonals (the maximum angle is taken). A square will

then show a skew angle of 90 degrees, while a distorted element will show angles over 150 degrees. The default tolerance is set to 110 degrees.

Wrap

This test is applied to quadrangle elements (quad4, quad8, quadb). It measures the coplanarity of the 4 vertices by comparing the height of the 4th point to the plan generated by the first three points (H), relatively to the element dimension. The exact formulation is $W = \frac{H}{l(D_1)+l(D_2)}$. Perfectly planar elements will have a null wrap coefficient. The default tolerance is set to 10^{-2}

RadiusEdge

This test is applied to tetrahedron elements (tetra4, tetra10). It measures the ratio between the radius of the circumsphere to the minimum edge length of a tetrahedron. Well shaped elements will show a small value while badly shaped elements will show far greater values. The radius edge coefficient is lower bounded by the radius edge ratio of the regular tetrahedron: $RE \geq \frac{\sqrt{6}}{4}$. The default maximum value is set to 2, which usually is sufficient to have a quality mesh. Sliver elements may not be detected by this measure.

Sliver

This test is applied to tetrahedron elements (tetra4, tetra10). A sliver element is a nearly flat tetrahedron, such pathology can lead to bad conditioning due to the very small volumes that can be engendered by these particular elements. This is well detected by computing the ratio between the maximum edge length to the minimum altitude (from a vertice to the opposed face). Sliver elements will have large values and possibly infinite if degenerated. The degenerated elements are set to a value of 10^5 for visualization, the default tolerance is set to 10.

FaceAspect

This can be applied to hexahedron and pentahedron elements (hexa8, hexa20, penta6, penta15). It measures the aspect ratio of each face of the elements. The default tolerance is set to 2.

Unstraight

This can be applied to any element with middle nodes. It measures the Euclidean distance between the edge middle (if the edge were straight) and the actual position of the middle edge node. Tolerance is set at 0.1.

RadiusCircum

This measure can only be accessed separately, with an explicity specification in the meas command. It measures the circum radiuses of triangle elements. This is applicable to tria3 and tria6 elements.

Commands

lab[...]

Outputs or prints the tests available and their default tolerance. If no output is asked this is printed to the prompt. `fe_quality('lab')` outputs the list of element tested with the command for detailed information. `fe_quality('lab EltName'` prints the tests available for the element *EltName* and the default tolerances associated.

meas[...]

Computes the mesh quality measurements. For a feplotmodel, the results are stored in the stack under the entry `'info','MeshQual'`. The results are given by element groups unless a specific element selection is given as a third argument. Accepted calls are `MQ = fe_quality('meas',model);` Computes all available tests per element group.

`MQ = fe_quality('meas -view MName',model);` Computes the *MName* test and visualize it.

`MQ = fe_quality('meas',model,'EltSel')` Computes all measurement tests for the specified *EltSel* element selection.

MQ is the mesh quality output. It is a structure of fields `eltid`, `data` and `lab`. All fields are cell arrays of the same size related to the measures described in the lab entry as `MName_ElemF_EGID` for which corresponding `EltId` and measurement values (`data`) are given. Direct visualization of the results can be obtained with the `-view` option.

view[...]

Performs a visualization of the quality measurements of a feplotmodel. The stack entry `'info','MeshQual'` must exist (created by meas). Two feplotselections are generated. First the elements are face colored in transparency with a colored ranking. Second, the elements outside the measurement tolerance are plotted in white patches of full opacity. Both plots generate an `EltSet`, the elements plotted are stored in `'set','MeshQual_eltsel'`, the elements outside tolerance are stored in `'set', 'MeshQual_MName_tol_val'` with *MName* the test considered and *val* the tolerance value.

The tolerance can be defined using the command option `-tol val`. A positive (resp. negative) tolerance *val* defines pathologic elements over (resp. under) the threshold.

Command option `-noGlobalMesh` customizes the selection so that the global mesh in transparency is not displayed.

It is possible to plot a sub selection of the elements measured by specifying an `EltSel` as third argument. The curve `colordataelt` plot can also be output.

`fe_quality('view');` Default visualization, `AspectRatio` is plotted as it is available for every element.

`fe_quality('view MName -tol val',cf);` `feplot`pointer, *MName* and tolerance *val* test are specified.

`fe_quality('view',cf,EltSel);` An additional element selection `EltSel` to restrict the mesh quality measurement plot.

MeshDim

`fe_quality('MeshDim',model)` returns a line vector `[weight average min max]` giving an indication on the mesh dimensions. The mesh edge lengths of all elements are computed, and the average, min and max data are output.

Command option `-print` allows printing this data in a human readable format to the output display.

Another use of command `MeshDim` is to recover element indices with a threshold on their length. Use command option `-getOverval` with a three output argument call. `[r1,r2,r3]=fe_quality('MeshDim -getOver val',model);` will output in `r3` element indices in `model.Elt` that verify

- minimum edge length over *val* if *val*>0
- miminum edge length under or equal to `abs(`*val*`)` if *val*<=0

print

Prints out the mesh quality report sorted in `'info', 'MeshQual'` of a model or a `feplot`figure. By default the results are printed to the prompt, a specific file can be given in the print command. *E.g.*

`fe_quality('print myMeshQualityReport',model);`

`CleanNJStraight`

This command attempts to improve a model numerical conditioning by straightening the edges of quadratic elements with negative Jacobians. This command can be iteratively performed as the local movements of specific middle edge nodes can result in distubances in the connected elements.

`model=fe_quality('CleanNJStraight',model);` will output the model with altered nodal positions corresponding to the edge straightening of quadratic elements with negative Jacobian.

Command option `-nit`*`N`* will ask to run a maximum of *`N`* passes unless all Jacobians become positive.

`clear[...]`

This command clears the element quality visualization and can also clean up the stack of any element sets created during the `view` procedures. All entries created by `fe_quality` in the model Stack are of the `'info'` or `'set'` type with a name starting by `MeshQual`.

`fe_quality('clear')` clears the `feplot`selection and visualization.

`fe_quality('clearall')` clears the visualization and removes every stack entry concerning mesh quality.

`fe_quality('clear `*`MName`*`')` removes from the stack a specified *`MName`* measurement visualization.

fe_range

Purpose

fe_range commands are used to manipulate experiment (series of design points) specifications.

Description

Experiments (series of design points) are used extensively in SDT. The figure below describes a 3 D design space with selected points. fe_range is used to generate experiment descriptions fe_range Build, run the solutions fe_range Loop and manipulate the associated results fe_range DirScan.

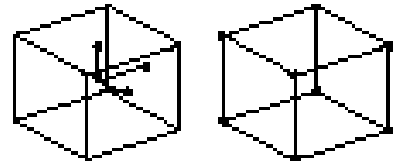


Figure 10.3: Sample experiments. a) Hypercube face center. b) Classical 2^{NP} factorial plan.

A range structure is the description of a set design points through a data structure with fields

- .val numeric array containing one design point per row and one design parameter per column.
- .lab cell array of strings giving a parameter label for each column. These labels should be acceptable fieldnames (no spaces, braces, ...)
- .param optional structure with fields associated with parameter labels used for formatting and analysis. Accepted values are detailed below. It is not necessary to define a .param field for each design parameter.

  param.MainFcn={FcnHandle,'command'} can be used specify the user handling function in fe_range Loop.
- .edge optional connectivity matrix used to define lines connecting different design points of the experiment
- .FileName optional cell array of strings used to build file names associated with each experiment with command fe_range fname.

.param fields must match string values in .lab. Each field is a struct with possible fields

- `.type` a string. Typically `double` or `pop`.
- `.choices`, for `.type='pop'`, contains a cell array of strings. The parameter value then gives the index within `.choices`.
- `.data` possible cell array containing data associated with the `.choices` field.
- `.LabFcn` a command to be evaluated with `st1=eval(r2.LabFcn)` to generate the proper label. For example `'sprintf(''%.1f ms'',val/1000)'` is used to generate a label in a different unit. For choices, the default is `r2.choices{val};`.
- `.Xlab` long name to be used to fill `Xlab` when generating curve data structures.
- `.level` is an integer specifying the computational step at which a given parameter can be modified. This is used to generate tree type experiments.
- `.uProp` is a cell array giving a coefficient to go from value to engineering unit and a string for the unit. Or a structure with fields `.coef` multiplicative coefficient to go from storage value in `val` to display value. `.coef` can be a callback string based on the assumption that the value is in a `val` variable. `.Xlab` to be used for display after unit conversion, `.unit` string for unit of storage value. `.fmt` java formatting for display in tables.
- `.SetFcn={fun,command}` provides the callback used in `fe_range Loop` to execute parameter setting.
- `.RepList={tokenString,subsasgn}` optionally provides the mechanism to aggregate parameter modifications in a `Loop` step.

Commands

`Build[R,stra]`

`Build` commands handle generation of the `Range` structure.

`Range=fe_range('Build'',par);`
`Range` is defined by a grid of all the parameter values defined in `par`.

`par` is expected to be either

- a structure where each field will be a parameter (for example `struct('a',(1:3)','b',(1:2)')`. For configurations, the field can contain a cell array with the configuration name and associated data (for example `struct('MesCfg',{'a',data1;'b',data2})`. To allow easier connection with graphical interfaces, the value can be replaced by a structure `struct('a',struct('Eval','1:3'))`. Note that `.param` is a reserved field.

- a cell array defining parameters. Each cell can be
  - a range data structure. This is for example used for visco parameter definitions (see `fevisco Range` for more details).
  - (to be phased out) a string *'lab "label" min min max max cur cur scale "scale" NPoints NPoints'*. *"label"* is the parameter name. Then the minimum, maximum and nominal values are defined. Scale can be "lin" for linear scale or "log" for logarithmic scale. *NPoints* defines the number of point for the parameter vector.
  - (to be phased out) a numeric vector in the old `upcom` format `[type cur min max scale]` with `type` defining the matrix type (unused here), `scale==2` indicates a logarithmic variation.

By default a grid type is generated. As an illustration, following example defines a grid 6x7 of 2 parameters named length and thickness

```
Range=fe_range('BuildGrid',struct('length',1:3, ...
   'thickness',[1 2],'Name',{{'a','data_for_a';'b','data_for_b'}}))
Range=fe_range('BuildGrid',Range);
fe_range('Tree',Range);

% String format (to be phased out)
par={'lab "length" min 10 max 20 cur 10 scale "lin" NPoints 6',...
     'lab "thickness" min 1e-3 max 2e-3 cur 0 scale "log" NPoints 7'};
Range=fe_range('BuildGrid',par);
fe_range('Tree',Range);
```

A `Range` type must be defined by token *stra*. The following strategies are supported

- `Grid` Generates a grid type, with all possible parameter combinations.
- `Vect` generates a vector for a single parameter or a matrix with all parameters varying (the initial definition of each vector must have the same length).
- `Simple` Generates a sequential parameter combination. One parameter varies at a time, the others being kept at their nominal value.
- `R`*stra* Generates reduced ranges spannix the parameters simplex. If ommitted this token is set to `MinMax` the available strategies are
  - *MinMax* base point with all max, then successive set of one parameter to min. Dedicated variants include

    - *MinMaxP* to add the nominal configuration at the list end
    - *MinMaxm* to add the full minimal configuration at the list end
  - *MinNom* uses the first design point with the nominal values of each parameters, then variations with one parameter set to the minimum value while the other remain at their nominal value.
  - *NomMax* uses the first design point with the maximum values of each parameters, then variations with one parameter set to the nominal value while the other remain at their maximum value.
  - *NomMin* uses the first design point with the minimum values of each parameters, then variations with one parameter set to the nominal value while the other remain at their minimum value.
  - *MaxMin* uses the first design point with the minimum values of each parameters, then variations with one parameter set to the maximum value while the other remain at their minimum value.
  - *MaxNom* uses the first design point with the nominal values of each parameters, then variations with one parameter set to the maximum value while the other remain at their nominal value.
  - *Nom* only uses the nominal value of each parameters.
  - *Grid* generates a grid type, considering minimum and maximum values of each parameter (corners of a hypercube).

  Command option `-flip` allows generating a reverted list from down to bottom (`flipud`).

- `@cbk` Allows customized defintion, using *cbk* as a function handle. `Range` is sequentially built with a expansion applied for each parameter. The typical call at the `j1`-th parameter is

  ```
  Range.val=cbk(Range.val,par{j1}.val,i1,RO);
  ```

  with `Range.val` the current `Range` before handling the `j1`-th parameter, `par{j1}.val` are the values considered for the `j1`-th parameter, `i1` is the column index associated to the `j1`-th parameter in `Range.val`. `RO` is the current running option structure.

Accepted options are

- `replace` ¿0 keep first value. `3` Reuse parameters in range.
- `level` handles levels if present.
- `FileName` augments the `Range.FileName` field if present.

- diag in Grid type, takes the diagonal of the hypercube.

```
Range=fe_range('BuildVect',par);
```

Simply concatenate all parameter ranges (they must have the same length) into a functional Range. par has the same format than for the fe_range BuildGrid input. In addition, all par entries provided should have the same number of points.

Vect command is used to generate single par structures to feed Range.param entries.

```
par={'lab "length" min 10 max 20 cur 10 scale "lin" NPoints 7',...
     'lab "thickness" min 1e-3 max 2e-3 cur 0 scale "log" NPoints 7'};
Range=fe_range('BuildVect',par);
```

DirScan

Scans a directory mat files and provides displayable information about property variations. It is assumed that files are saved with a variable RO (for Run Options) in struct format, each option considered as a field. Command DirScan will build a synthesis between constant and variable options, by providing in output a structure RB with fields

- wd the scanned directory.
- dirlist the file names scanned. (By default *.mat).
- RVar a structure that can be displayed by comstr -17 in Java tabs. In its basic version it contains fields table that list the differing options between each scanned file, one per line, and a field ColumnName that provides the relative varying option fields. This list is handled by the flatParamFcn as explained below.
- RConstant a cell array with two columns providing the constant options for all scanned files. The first column contains the option fields and the second their value.

By default DirScan saves a file named RangeScan.mat in the scanned directory. This file contains the output to avoid scanning if possible. By default scanning is skipped if the file RangeScan.mat exists, refers to the same search in the dirlist field and if this file is more recent than all files to be scanned.

Options sorting is performed by the flatParamFcn. Typical options are hierarchically sorted in nested structure format that gather parameters of the same type, or belonging to a configuration set.

To ensure a clean view of varying parameters, the hierarchical structure has to be flattened, that is to recursively flush back all nested structure fields to the root structure. The default `flatParamFcn` only performs this simple operation, one should not use identical parameter names in different locations.

For advanced applications it is recommended to add intelligence to the `flatParamFcn` to help sorting relevant parameters, possibly remove some irrelevant ones and to convert complex options into human readable format. The typical call to `flatParamFcn` is

```
r1=feval(flatParamFcn,fname,RO);
```

The output `r1` is in the same format than output `RConstant` field, that is a two column cell array with fieldnames in first column defining found parameters and a second column containing current values for the currently scanned file.

Input `fname` is a structure with field `fname` providing the file name containing a parameter structure names `RO`. Input `RO` is a structure with fields `wd` providing the name of the scanned directory and `Content`, a cell array that will keep track of all parameter fieldnames encountered during scanning.

A sample call would be

```
% Example of flatParamFcn behavior
% build a dummy result file with a parameter structure
tname=nas2up('tempname_RES.mat');
% sample 2 level parameter structure
RO=struct('MeshCfg',...
struct('lc',4,'name','toto'),...
'SimuCfg',...
struct('dt',1e-2','Tend',10));
% save RO in result file
save(tname,'RO');
% Options to call flatParam
R1=struct('flatParamFcn',fe_range('@flatParam'),'wd',pwd,'Content',{{}});
% call to flatParam
[r1,RO]=feval(R1.flatParamFcn,struct('fname',tname),R1);
delete(tname); % clean up example
```

Command `DirScan` takes a structure in input with fields

- `wd` the directory to scan.
- `list` the file names to scan. This allows restriction to filenames matching specific expressions. The default is `*.mat`.

- flatParamFcn. A function handle to sort the running options, by defaultset to fe_range('@flatParam').
- NoSave not to save the result in RangeScan.mat if set to one.

The following command options are accepted

- -reset to force a new scanning.
- -reload to force a reload of RangeScan.mat.

fname [,LabCell,Labdef]

Generate the list of files using the components of the Range.FileName cell array. In that cell, each string starting with '' is replaced. fnamedir forces names compatible with directories.

```
Range=fe_range('BuildGrid',struct('length',1:3, ...
   'thickness',[1 2],'Name',{{'a','data_for_a';'b','data_for_b'}}))
Range=fe_range('BuildGrid',Range);
% Name built out of differnt labels
Range.param.thickness.LabFcn='sprintf(''h=%.1f'',val)';
Range.param.thickness.ShortFmt=1;
Range.FileName={'Root','@length','@thickness','@Name'};
fe_range('fname',Range)
```

GeneLoop

Provides a genetic algorithm implementation inspired from the NSGA-II [50].

Command GeneLoop peforms the complete loop, taking into argument a set of parameters in a cell array, and a parameter structure with fields

- .PopSize the population size,
- .MatSize the mating group size,
- .NbTour the number of candidates to retain for a tournament round,
- .RatioMut the threshold ratio under which a gene mutation occurs when there is no crossover, between 0 (impossible) and 1 (always)

- .Optim the extremum to consider for the fitness function, min or max
- .RunExp the callback to the fitness function

The logic is to exploit a discretized pool gene based on a Range structure with available parameters. From a randomly chosen initial population using randi of size .PopSize, individuals are selected for mating, based on their fitness in a tournament phase. The tournament consists in running .MatSize tours in which the fittest individual is taken between .NbTour randomly picked candidates. The .MatSize selected individuals are then mated with a crossover and mutation strategy to produce a children gene pool of size PopSize. Crossover and mutation are sequential events. First, a crossover generates one child from two randomly picked parents in the mating pool (based on randperm) each gene is randomly picked from one or the other parent. Each gene can then mutate with a probability event driven by .RatioMut (based on rand threshold). In case of a mutation event, the gene will be forced to mutate by taking another available value in the gene pool. The children gene pool is forced to be gene combinations that have not been tested before. The parent and chidren populations are then combined, and only the .PopSize fittest individuals are kept for the next generation, ensuring elitism.

The output is a Range structure with field .val containing the current population, .Res the fitness values of the current population, .val0 the archive of all tested individuals, .Res0 the archive of fitness values of all tested individuals.

```
% define a set of parameters with discretized varying values
par={'lab "p1" min 10 max 20 cur 10 scale "lin" NPoints 100',...
  'lab "p2" min 1 max 100 cur 1 scale "lin" NPoints 100',...
  'lab "p3" min 2 max 5 cur 100 scale "log" NPoints 100'};

% define a callback function updating Range.Res
% with fitness function based on Range.val
RunExp=@(x,~)setfield(x,'Res',abs(sqrt(x.val(:,1)+x.val(:,2).^3)-x.val(:,3)));

% Define genetic algorithm options
RO=struct('MaxGen',100,'PopSize',25,'MatSize',10,'NbTour',4,...
 'RatioMut',0.4,'RunExp',RunExp,'Optim','min');

% Run genetic algorithm
Range=fe_range('GeneLoop',par,RO);
```

labFcn

Loop

Loop the generic handler of parametric studies.

- The outer loop performs a loop on rows of Range.val (design points)
- For each design point, a loop on levels is performed. At a given level, an action is performed if .param.lab.SetFcn={'FcnName','Command'} is defined. When aggregating parameters of a given level (typically with a LabCfg parameter), it is expected that only the configuration parameter has a SetFcn and fe_range ValEvtMerge is called.
- Range.param.MainFcn={'FcnName','Command'} is used to implement standard methods.

Standard calls are:

```
fe_range('Loop',Range,RO)
fe_range('Loop',Range,UI,RO)
```

RO is

- .restart do a restart.
- .StepStart defines the minimum step level to be computed. If omitted starts at 1.
- .nSteps defines the maximum step level to be computed. If omitted, all steps are computed.
- .Verbose define the verbosity level. 0 by default.
- .WaitAt can be used to generate wait files at a given step number. It is then possible to open MATLAB slaves that will consume waiting files present in a given directory by calling sdtjob('StudySlaveStart reset',pwd). This provides a simple mechanism for parallel execution of a series of steps. It is however then expected that the step saves its results to a file that will not interact with other jobs.

Res

```
R1=fe_range('Res',R1,Range);
```

This command reshapes the last dimension of the result curve R1 according to the Range. For a grid DOE last dimension is split in as many dimensions as parameters. For a vector DOE, last dimension is only redefined by a cell array of labels defining each design point.

The following command options are available

- -varOnly to expand only varying parameters in Range. In such case, constant parameters are gathered in the last dimension of label RConst.
- -var*name* to only extract parameter *name* from Range. For this to be possible, this parameter must be gridded against the remaining ones.
- -noRConst not to keep dimension RConst when using -varOnly
- -noParLab not to forward parameter LabFcn that possibly exist in Range to the .Xlab curve entry.

Sel

This command allows selection of design points in a series of experiments described by a Range structure. The main output is the indices in Range.val rows corresponding to the sequential application of selection rules.

The selection rules a provided in a cell array of three columns and as many lines as rules to apply under the format
{param_name,'rule','crit';...}.
The following types of rules are supported, defined by a string,

- ismember applies selection by only taking the values specified using MATLAB ismember command. crit is then either a list of values (then corresponding to values appearing in the DOE table), or a cell array of values (then corresponding to the values in the DOE table where string values are used for pop style parameters. Regular expressions are supported for the pop entries, in which case the string must start by # followed by the regular expression to apply.
- <,>,<=,>=,== applies sampling by using the logical operator specified on the parameter values. crit is then a numerical value corresponding to the values appearing in the DOE table for all parameters.
- sort applies a sorting algorithm for a given parameter. crit is then either
  - a string specifying an argument to the sort command of MATLAB, either ascend or descend. Support for pop types is provided based on alphabetical sorting.
  - a function_handle to a sorting function that will be called with the val or choices field of the parameter and that will rethrow the sorted values and the corresponding index to the unsorted values.

  - a cell array callback with first field a function_handle that will be called, the second entry will be replaced by the val or choices field of the parameter, and any further entries provided.

- sortrows will perform a post-treatment of the sampled Range to the selection applied and output a java compatible table.

Excepted for sortrows, other rules are sequentially applied to the current sampled Range. Sorting is thus only fully effective if last performed.

The optional .SortCol field can be used to specify a reformatting of the indices as a multi-dimensional grid.

Stats

```
fe_range('Stats',UI,sel,RA);
```

This command can be used to call a stack of post-treatments for a subpart of all computation results that have been priorly scanned through the DirScan command, and then displayed in the RVar tab. Results of the Stats command is a Stats tab in the UI.

UI is the interface data (where the Stats tab will be displayed), that can be obtained through the *MainFcn*('ParamUI') command. If it is left empty (UI=[]), sdtroot interface is implicitely defined.

sel is a selection cell array to select a sub set of results in all scanned results. See Sel for more details. If sel is empty (sel={}), all results are post-treated.

RA defines the post-treatments to be computed from selected results. It is a data structure with following fields:

- .SortCols defines a subset and the order of input parameters to consider for stats output or display. Parameters must be RVar parameters.

- .PostPar is a data structure with a .list field that defines the stack of post-treatments to be computed. PostPar.list is a cell array with as many rows as post-treatments to compute. Each row is of the form 'PostName' PostData. PostData is a cell array of the form {cbk data}.

  - cbk is callback cell array of the form {FcnHandle Arg1 Arg2 ...}. The callback is called with [Full,Stat]=feval(cbk{1},obj0,evt0,cbk{2:end});. obj0 contains the results read in the current mat result file and evt0 is a structure with evt0.j1 containing the indice in the results stack. There are 2 output arguments Full that should contain a

full signal (for example the observation of time deformation at sensors, ...) and Stats that should be a SDT curve table (1 or 2 dimensions) of scalar results (for example time statistics for time signals...).

- data is a cell array with as many rows as scalar results (1 by column in the Stats tab that will be displayed) to extract from the post-treatment result curve Stats. Each row is of the form i1 i2 CritFcn. i1 (resp. i2) can be either the row (resp column) indice of the scalar result in the Stats.Y table or the label of the row (resp column) in the Stat.X{1} abscissa (resp Stat.X{2}). CritFcn is the handle of a criterion function that can be used to colorate cell in the result tab (for example a threshold function) and also to display boundary lines in iiplot displays (needs critfcn doc and example).

Simple

Generates a set of experiments with sequential variation of each parameter, the other ones being fixed to their nominal value. par has the same format than for the fe_range BuildGrid input. They may feature a field nom providing a nominal value to each parameter, if this field is omitted the nominal value is considered to be the starting value of the parameter. In the case where par has been defined as a string input, field nom is taken to be the cur input value.

```
par={'lab "length" min 10 max 20 cur 10 scale "lin" NPoints 6',...
     'lab "thickness" min 1e-3 max 2e-3 cur 0 scale "log" NPoints 7'};
Range=fe_range('Simple',par);
```

UI Tree

Basic display of an experiment design as a tree. See also the sdtroot version.

```
par={'lab "length" min 10 max 20 cur 10 scale "lin" NPoints 6',...
     'lab "thickness" min 1e-3 max 2e-3 cur 0 scale "log" NPoints 7'};
Range=fe_range('Simple',par);
fe_range('Tree',Range);
sdtroot('setRange',Range); % Initialize range in PA.Range
PA=sdtroot('PARAMVh');PA.Range
sdtroot('InitRange'); % Initialize display
```

Val

Val commands are used to ease range manipulations.

`ValCell`

`r2=fe_range('ValCell',Range);`
This command can be used to convert a `Range.Val` as a cell array with as many rows as `Range.val` and each row of the form `param1, val1, param2 val2, ...`. One can give `ind` as a 2nd argument, with the indices of rows to convert.

fe_reduc

Purpose

Utilities for finite element model reduction.

Syntax

```
SE = fe_reduc('command options',model)
TR = fe_reduc('command options',model)
```

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 6.2). Input arguments are a command detailed later and a model (see section 7.6). Obsolete low level calls are detailed at the end of this section. Generic options for the command are

- -matdes can be used to specify a list of desired matrices. Default values are -matdes 2 1 for mass and stiffness, see details in section 4.8.8 .
- -SE is used to obtain the output (reduced model) as a superelement SE. Details about the fields of superelement data-structures are given section section 6.3.2 .
- model.Dbfile can be used to specify a -v7.3 .mat file to be used as database for out of core operations.
- -hdf is used to request the use of out of core operations.

When using a model with pre-assembled matrices in the .K field, boundary conditions must not be eliminated to avoid reassembly in fe_reduc which is indicated by the message Assembling model.

```
[SE,CE] = fe_case(model,'assemble -matdes 2 1 -SE -NoT');
SE=stack_set(SE,'case','Case 1',CE);
```

Accepted fe_reduc commands are

Static, CraigBampton

Static computes *static* or *Guyan condensation*. CraigBampton appends fixed interface modes to the static condensation.

Given a set of interface DOFs, indexed as I, and other DOFs C, the static responses to unit displacements are given by

$$[T] = \begin{bmatrix} T_I \\ T_C \end{bmatrix} = \begin{bmatrix} I \\ -K_{CC}^{-1} K_{CI} \end{bmatrix} \tag{10.19}$$

which is the static basis (also called *constraint modes* in the Component Mode Synthesis literature). For Craig-Bampton (6.107), one appends fixed interface modes (with $q_I = 0$). Note that you may get an error if the interface DOFs do not constrain rigid body motion so that K_{CC} is singular.

The interface DOFs should be specified using a `DofSet` case entry. The interface DOFs are defined as those used in the `DofSet`. The complementary DOF are determined by exclusion of the interface DOF from the remaining active DOFs.

```
model=demosdt('volbeam');
% Define interface to be xyz DOF at nodes 2,3
model=fe_case(model,'DofSet','IN', ...
    feutil('getdof',[2;3],[.01;.02;.03]));
% statically reduced model
ST=fe_reduc('Static',model);
% For Craig Bampton specify eigenvalue options
model=stack_set(model,'info','EigOpt',[5 10 0]);
CB=fe_reduc('CraigBampton',model);
```

Available command options are

- `NM` is the number of desired modes, which *should be specified* in an `info,EigOpt` stack entry which allow selection of the eigenvalue solver (default is 5, Lanczos). Note that using `NM=0` corresponds to static or Guyan condensation.
- `-SE` is used to obtain the output as a superelement SE. Without this argument, outputs are the rather obsolete list `[T,sdof,f,mr,kr]` where f is the frequency of fixed interface modes.
- `-shift` allows the use of a non-zero shift in the eigenvalue solution for the fixed interface modes. The interior matrix K_{cc} is only factored once, so using a shifted matrix may result in poor estimates of rigid body modes.
- `-useDOF` recombines the fixed interface modes to associate shape with a specific interior DOF. This can ease the manipulation of the resulting model as a superelement.
- `-drill`. Shell elements may not always use drilling stiffness (5 DOF rather than 6), which tends to cause problems when using 6 DOF interfaces. The option calls `model.il=p_shell('SetDrill 0',model.il)` to force the default 6 DOF formulations.

- -Load appends static correction for defined loads to the model.

mdl=fesuper(mdl,'setTR',*name*,'fe_reduc command') calls fe_reduc to assemble and reduce the superelement. For example
mdl=fesuper(mdl,'SetTR','SE1','CraigBampton -UseDof -drill');

Switching to out of core solver using .mat files is based on the value of sdtdef('OutOfCoreBufferSize') given in Mb. For a sufficiently large RAM, you may want to use larger values sdtdef('OutOfCoreBufferSize',1024*8) for 8 GB.

Free ...

The standard basis for modal truncation with static correction discussed in section 6.2.3 (also known as McNeal reduction). Static correction is computed for the loads defined in the model case (see fe_case). Accepted command options are

- *EigOpt should be specified* in an info,EigOpt stack entry. For backward compatibility these fe_eig options can be given in the command and are used to compute the modeshapes. In the presence of rigid body modes you must provide a mass shift.
- Float=1 is used to obtain the standard attachment modes (6.103) in the presence of rigid body modes. Without this option, fe_reduc uses shifted attachment modes (6.104), when a non zero shift is given in *EigOpt*. This default is typically much faster since the shifted matrix need not be refactored, but may cause problem for relatively large negative shifts.

  Float=2 uses an inertia balancing with respect to computed modes.
- -SE is used to obtain the output as a superelement SE.
- -bset returns information about loads to be applied in a system where enforced motion (fe_load DofSet) entries are defined.
- -FirstCB implements first order correction for damping terms associated with viscous or hysteretic damping.

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)

$$\left[K_{Flex}^{-1}\right][b] = \sum_{j=NR+1}^{N} \frac{\{\phi_j\}\{\phi_j\}^T}{\omega_j^2} \tag{10.20}$$

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
$$[q_{Iso}] = \begin{bmatrix} 0 & 0 \\ 0 & K_{Iso}^{-1} \end{bmatrix}[b_{Flex}] \tag{10.21}$$
- 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 [37] 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

$$[\phi_R] = \begin{bmatrix} \phi_{RR} \\ \phi_{IsoR} \end{bmatrix} = \begin{bmatrix} I \\ -K_{Iso}^{-1} K_{IsoR} \end{bmatrix} \tag{10.22}$$

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.

obsoletem,k,mdof (obsolete format)

Low level calling formats where matrices are provided are still supported but should be phased out since they do not allow memory optimization needed for larger models.

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.

For CraigBampton, the calling format was
fe_reduc('CraigBampton *NM Shift Options*',m,k,mdof,idof);.

See also

fe2ss, fe_eig, section 6.2

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 3.1.3 and the associated `d_cor('TutoSensPlace')` demo. `fe_sens` provides sensor/shaker placement methods.

InAcceptable

Command `InAcceptable` defines a set of acceptable sensors measuring normal displacement on a surface. This is typically used for hammer testing where duality/reciprocity is used to place sensors that are then used as impact locations (while shaker placement to define locations of reference accelerometers). Normal displacement is also typical for shaker placement.

Syntax is `model=fe_sens('InAcceptable',model);` with `model` a standard SDT model. The output `model` is the same model with a `SensDof` case entry named `Acceptable` containing the surface normal displacement observation, and a `FaceId` set named `AcceptableMap` containing the faces selected for the observation.

The main advantage of this command is the possibily to restrict the acceptable positions by using a third argument structure to possibly alter selection,

- `.sel` to provide a `FindElt` model selection on which the search is performed. By default a `selface` selection is performed.
- `.EdgeTol` to remove sharp edges. Shaker positioning is indeed impossible or difficult on nodes placed on sharp edges. Field `.EdgeTol` if present is a numeric value taken as an angle threshold in degrees. Facets showing angles on edges with others facets over the given threshold are eliminated.
- `.radius` to restrict the search to element groups (in the SDT terms, see `elt`) with a significant spatial span. If present, field `.radius` is a numeric threshold taken as the miminum sphere radius in which groups should not fit.

Example:

```
% Find acceptable position for shaker placement
demosdt('demoubeam'); cf=feplot; def=cf.def;
RO=struct('sel','selface', ...
 'EdgeTol',20);
model=cf.mdl.GetData; % Temporary model

model=fe_sens('InAcceptable',cf.mdl,RO);

% display model with new stack entries
feplot('initmodel',model)
% display added set
cf.sel='setname AcceptableMAP';
% display added SensDof entry
fecom(cf,'curtabCases','Acceptable');fecom(cf,'proviewon');
```

InGetDn

`model=fe_sens('InGetDn',model,def);` appends normal displacement associated with DOF `.19`.

`model` is a FEM `model`, `def` is a curve defined on the model. By default, this command expects to find the `SensDof` entry `Acceptable` generated by `fe_sens InAcceptable`. One can however provide a custom observation using a third agument `SensDofName` as string defining a `SensDof` entry in `model`.

By default the output is `model` with added stack entries

- `curve,InDef` providing the `def` structure with added observations on DOF `.19`.
- `info,SensDofName` providing the observation structure used. Note that `SensDofName` is here the name provided in third argument with default set to `Acceptable`.

The following command options are available

- `-def` outputs the `def` structure with added observations on DOF `.19` instead of `model`.
- `-d19` only generates the `def` output on DOF `.19`.

The follwing example is based on the pre-treatment performed by `fe_sens InAcceptable`.

```
% Add additional nodal observations to an existing curve
% Normal displacement appended for shaker placementp procedures
model=fe_sens('InGetDn',model,def);
d1=stack_get(model,'curve','InDef','get')
```

indep

`sdof=fe_sens('indep',DEF)` uses the effective independence algorithm [14] 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 `d_cor('TutoSensPlace')` demo. The `mseq` algorithm is much faster and typically gives better results.

mseq

`sdof = fe_sens('mseq` *`Nsens target`*`',DEF,sdof0)` places *`Nsens`* sensors, with an optional initial set *`sdof0`*. The maximum response sequence algorithm [49] 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`*.

```
[FEM,def]=demosdt('demo gartfe');
def=fe_def('subdef',def,6:15); % Keep ten modes
d1=fe_def('subdof',def,[.01;.02;.03]) % Keep translations
% Select subpart as target location
d1=fe_def('subdof',d1,feutil('findnode group 4:6',FEM));
sdof= fe_sens('mseq 10',def);
FEM=fe_case(FEM,'sensdof','Test',sdof);
feplot(FEM);fecom('curtabCase -viewOn','Test');
% see also  garsens 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. To get started you can refer to the following sections

- defining a wire-frame with translation sensors in section 2.7.1 and section 2.7.2
- adding sensors to a FEM as a SensDof entry is illustrated in the topology correlation tutorial section 3.1 .

Commands supported by fe_sens are

basis

These commands are used to handle cases where the test geometry is defined in a different frame than the FEM. An example is detailed in section 3.1.2 .

BasisEstimate guesses 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. Arguments are the model, and the name of the SensDof entry containing a test frame.

```
model = fe_sens('basisEstimate',model,'Test');
```

A list of node pairs in the FEM and test frames can be provided as an additional argument to improve results. The list is a two columns matrix containing FEM (resp. test) NodeId in the first (resp. second) column. If four nodes are provided, the estimation is an exact triplet positioning, the first node being the origin and the 3 other being directions (must be non collinear). For shorter or longer node lists, the positioning is based on global distance minimization between paired nodes.

BasisEstimate2 uses another strategy to guess a reasonnable superposition of the test wireframe over the FEM, based on finding main directions (SVD of the node sets) and their corresponding orientations. It should be used rather than BasisEstimate when it is not expected that global coordinates (X,Y,Z directions) of both the test and the FEM coincide.

Basis is used to set the local test basis in a script (see example in section 3.1.2). Once the script is set, command option -noShow allows not printing the setting script to the screen.

BasisToFEM is used to transform the SensDof entry to FEM coordinates. This transformation is done after basis adjustment and makes verification easier by clarifying the fact that the sens.tdof uses the 5 column format with measurement directions given in the FEM format. The only reference to test is the identifier in sens.tdof(:,1) which is kept unchanged and thus where a 1.01 will refer to test direction x which may be another direction in the FEM.

SensMatch, sens, ...

For the basic definition of translation sensors is associated with cell arrays giving {'SensId','x','y','z','DirSpec'}, as detailed in section 4.6.2 .

The building of observation matrices for SensDof entries is now described under sensor SensMatch (building topology correlation to locate test nodes in the FEM model) and sensor Sens (building of the observation matrix after matching). Please read section 4.6.4 for more details.

The obsolete near,rigid,arigid commands are supported through SensMatch calls.

tdof, ...

tdof = fe_sens('tdof',sens.tdof) returns the 5 column form of tdof if sens.tdof is defined as a DOF definition vector. For more details see sens.tdof, section 2.7 for test geometry definition, and section 4.6 for general sensor definitions in FEM models.

sens=fe_sens('tdoftable',tcell,sens); is used to generate a group of sensors from a table a illustrated in section 4.6.2 . The sens may be omitted of all the information is given in the table. The command option InFEM is used to generate sensors that use FEM degree of freedom.

fe_sens('tdoftable',model,'SensDofEntry'); is used to generate the table description of the given group of sensors (with no output argument, the table is displayed).

links

fecom('ShowLinks Sensors') generates a plot with the mode wire-mesh associated with the SensDof entry Sensors.

For older models where the wire frame is included in the model with a negative EGID, fecom('ShowLinks') still 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.

WireExp

def = fe_sens('wireexp',sens) uses the wire-frame topology define in sens to create an interpolation for un-measured directions. For a tutorial on this issue see section 3.3.2 .

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 (the drums of the green deformation have global motion rather than just one point moving horizontally).

```
[TEST,test_mode]=demosdt('demo gartte wire');
TR=fe_sens('wireexp',TEST);
cf=feplot;cf.model=TEST;fe_sens('WireExpShow',cf,TR)
pause %Use +/- to scan trough deformations as a verification

cf.def(1)=test_mode;
cf.def(2)={test_mode,TR};
fecom(';show2def;ScaleEqual;ch8;view2');
legend(cf.o(1:2),'Nominal','Wire-exp')
```

The command builds default properties associated with the wire frame (beams properties for segments, shells properties for surfaces, elastic properties for volumes). In some cases you may get better properties by defining properties yourself (see section 7.4 and section 7.3).

Test mesh handling

fe_sens provides commands dedicated to test mesh manipulations prior to correlation.

MeshProject

TEST=fe_sens('MeshProject',TEST,'x',[x1 x2 x3],...); Projects the test mesh from a basis declaration. This allows keeping a test mesh in a specific basis for reuse in different FEM. Usual options are x, y, origin, scale. The command then defines a basis comptatible with the input arguments and projects nodes and tdof in the mesh output.

```
TEST=fe_sens('MeshProject',TEST,...
{'x',    [-0.0724899 0.0460858 -0.996304], ... % x_test in FEM coordinates
 'y',    [-0.0846775 0.995041 0.0521884], ... % y_test in FEM coordinates
 'origin',[182.78955815663335 25.33640608720782 -1.0],... % test origin in FEM coor
 'scale', [1]};
```

MeshSub

T1=fe_sens('MeshSub',TEST,NodeSel); Generates a test mesh T1 from a node selection from the global test mesh TEST. Remaining edges of cut surface elements will be kept as beams.

Section 4.6, 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,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 4.8 .

Once you have defined a FEM model (section 4.5), material and elements properties (section 4.5.1), loads and boundary conditions (see fe_case), 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 4.8) on how to compute a response.

Input arguments are :

- MODEL a standard FEM model data structure with loads, boundary conditions, ... defined in the case. See section 4.5 (tutorial), fe_case for boundary conditions, fe_load for loads, ...
- OPT is an option vector or data structure used for some solutions. These may also be stored as model.Stack entries.

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,'FVol','Volume load',data);
[cf.def,model]=fe_simul('static',model);
```

- 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;
model=stack_set(model,'info','EigOpt',[6 10 0 11]);
[cf.def,model]=fe_simul('mode',model);
```

- DFRF: computes the direct response to a set of input/output at the frequencies defines in Stack.

```
femesh('reset'); model = femesh('testubeamt');
model=fe_case(model,'FixDof','Clamped end','z==0');
r1=struct('DOF',365.03,'def',1.1); % 1.1 N at node 365 direction z
model=fe_case(model,'DofLoad','PointLoad',r1);
model= stack_set(model,'info','Freq',1:10);
def=fe_simul('DFRF',model);
```

  One can define a frequency dependence of the load using a curve (see section 7.9 for more detail). For example:

```
model=fe_curve(model,'set','input','Testeval (2*pi*t).^2');
model=fe_case(model,'setcurve','PointLoad','input');
```

- Time : computes the time response. You must specify which algorithm is used (Newmark, Discontinuous Galerkin dg, Newton, Theta, or NLNewmark). For transient computations, opt=[beta alpha t0 deltaT Nstep] (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,[.25 .5 0 1e-4 50]);
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_stress

Purpose

Computation of stresses and energies for given deformations.

Syntax

```
Result = fe_stress('Command',MODEL,DEF)
  ...  = fe_stress('Command',node,elt,pl,il, ...)
  ...  = fe_stress( ... ,mode,mdof)
```

Description

You can display stresses and energies directly using `fecom ColorDataEner` commands and use `fe_stress` 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 data structure 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`.

`Ener [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. Accepted command options are

- `-MatDes`*`val`* is used to specify the matrix type (see `MatType`). `-MatDes 5` now correctly computes energies in pre-stressed configurations.
- `-curve` should be used to obtain energies in the newer curve format. `Ek.X{1}` gives as columns `EltId,vol,MatId,ProId,GroupId` so that passage between energy and energy density can be done dynamically.
- `ElementSelection` (see the element selection commands) used to compute energies in part of the model only. The default is to compute energies in all elements. A typical call to get the strain energy in a material of ID 1 would then be `R1=fe_stress('Ener -MatDes1 -curve matid1',model,def);`

Obsolete options are

- m, k specify computation of kinetic or strain energies. For backward compatibility, fe_stress returns [StrainE,KinE] as two arguments if no element selection is given.
- dens changes from the default where the element energy and **not** energy density is computed. This may be more appropriate when displaying energy levels for structures with uneven meshes.
- Element energies are computed for deformations in DEF and the result is returned in the data structure RESULT with fields .data and .EltId which specifies which elements were selected. A .vol field gives the volume or mass of each element to allow switching between energy and energy density.

The strain and kinetic energies of an element are defined by

$$E^e_{strain} = \frac{1}{2}\phi^T K_{element}\phi \quad \text{and} \quad E^e_{kinetic} = \frac{1}{2}\phi^T M_{element}\phi$$

For complex frequency responses, one integrates the response over one cycle, which corresponds to summing the energies of the real and imaginary parts and using a factor 1/4 rather than 1/2.

feplot

feplot allows the visualization of these energies using a color coding. You should compute energies once, then select how it is displayed. Energy computation clearly require material and element properties to be defined in InitModel.

The earlier high level commands fecom ColorDataK or ColorDataM don't store the result and thus tend to lead to the need to recompute energies multiple times. The preferred strategy is illustrated below.

```
% Computing, storing and displaying energy data
 demosdt('LoadGartFe'); % load model,def
 cf=feplot(model,def);cf.sel='eltname quad4';fecom ch7
 % Compute energy and store in Stack
 Ek=fe_stress('ener -MatDes 1 -curve',model,def)
 cf.Stack{'info','Ek'}=Ek;
 % Color is energy density by element
 feplot('ColorDataElt  -dens -ColorBarTitle "Ener Dens"',Ek);
 % Color by group of elements
 cf.sel={'eltname quad4', ... % Just the plates
   'ColorDataElt -ColorBarTitle "ener" -bygroup -edgealpha .1', ...
   Ek}; % Data with no need to recompute
 fecom(cf,'ColorScale One Off Tight') % Default color scaling for energies
```

Accepted ColorDataElt options are

- -dens divides by element volume. Note that this can be problematic for mixed element types (in the example above, the volume of celas springs is defined as its length, which is inappropriate here).
- -frac divides the result by the total energy (equal to the square of the modal frequency for normal modes).
- -byGroup sums energies within the same element group. Similarly -byProId and -byMatId group by property identifier. When results are grouped, the fecom('InfoMass') command gives a summary of results.

The color animation mode is set to ScaleColorOne.

Stress

out=fe_stress('stress *CritFcn Options*',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).
-comp *i*	Returns the stress components of index *i*. This component index is giving in the engineering rather than tensor notation (before applying the TensorTopology transformation).

Supported command Options (to select a restitution method, ...) are

- AtNode average stress at each node (default). Note this is not currently weighted by element volume and thus quite approximate. Result is a structure with fields .DOF and .data.
- AtCenter stress at center or mean stress at element stress restitution points. Result is a structure with fields .EltId and .data.

- AtInteg stress at integration points (*b family of elements).

- Gstate returns a case with Case.GroupInfo{jGroup,5} containing the group gstate. This will be typically used to initialize stress states for non-linear computations. For multiple deformations, gstate the first nElt columns correspond to the first deformation.

- -curve returns the output using the curve format.

The fecom ColorDataStress directly calls fe_stress and displays the result. For example, run the basic element test q4p testsurstress, then display various stresses using

```
% Using stress display commands
 q4p('testsurstress')
 fecom('ColorDataStress atcenter')
 fecom('ColorDataStress mises')
 fecom('ColorDataStress sII atcenter')
```

To obtain strain computations, use the strain material as shown below.

```
% Accessing stress computation data (older calls)
 [model,def]=hexa8('testload stress');
 model.pl=m_elastic('dbval 100 strain','dbval 112 strain');
 model.il=p_solid('dbval 111 d3 -3');
 data=fe_stress('stress atcenter',model,def)
```

CritFcn

For stress processing, one must often distinguish the raw stress components associated with the element formulation and the desired output. CritFcn are callback functions that take a local variable r1 of dimensions (stress components × nodes × deformations) and to replace this variable with the desired stress quantity(ies). For example

```
% Sample declaration of a user defined stress criterium computation
 function out=first_comp(r1)
  out=squeeze(r1(1,:,:,:));
```

would be a function taking the first component of a computed stress. sdtweb fe_stress(''Principal'') provides stress evaluations classical for mechanics.

For example, a list of predefined CritFcn callback :

- Von Mises : `CritFcn='r1=of_mk(''StressCrit'',r1,''VonMises'');lab=''Mises'';';`
- YY component : `CritFcn='r1=r1(2,:,:,:);lab=''Syy'';'`

Redefining the `CritFcn` callback is in particular used in the `StressCut` functionality, see section 4.7 .

See also

`fe_mk`, `feplot`, `fecom`

fe_time

Purpose

Computation of time and non linear responses.

Syntax

```
def=fe_time(model)
def=fe_time(TimeOpt,model)
[def,model,opt]=fe_time(TimeOpt,model)
model=fe_time('TimeOpt...',model)
TimeOpt=fe_time('TimeOpt...')
```

Description

`fe_time` groups static non-linear and transient solvers to compute the response of a FE model given initial conditions, boundary conditions, load case and time parameters. Note that you may access to the `fe_time` commands graphically with the simulate tab of the feplot GUI. See tutorial (section 4.8) on how to compute a response.

Solvers and options

Three types of time integration algorithm are possible: the Newmark schemes, the Theta-method, and the time Discontinuous Galerkin method. Implicit and explicit methods are implemented for the Newmark scheme, depending on the Newmark coefficients β and γ, and non linear problems are supported.

The parameters of a simulation are stored in a time option data structure `TimeOpt` given as input argument or in a `model.Stack` entry `info,TimeOpt`. Initial conditions are stored as a `curve,q0` entry.

The solvers selected by the string `TimeOpt.Method` are

- `newmark` linear Newmark
- `NLNewmark` non linear Newmark (with Newton iterations)
- `staticNewton` static Newton
- `Theta` Theta-Method (linear)
- `Euler` method for first order time integration.

- dg Discontinuous Galerkin
- back perform assembly and return model,Case,opt.

Here is a simple example to illustrate the common use of this function.

```
model=fe_time('demo bar'); % build the model

% Define time options and use structure directly
opt=fe_time('TimeOpt Newmark .25 .5 0 1e-4 100');
def=fe_time(opt,model); % compute the response

% Store as model.Stack entry {'info','TimeOpt',opt}
model=stack_set(model,'info','TimeOpt',opt);
def=fe_time(model); % compute the response
```

TimeOpt

The TimeOpt data structure has fields to control the solver

- Method selection of the solver
- Opt numeric parameters of solver if any. For example for Newmark [beta gamma t0 deltaT Nstep]
- MaxIter maximum number of iterations.
- nf optional value of the first residual norm. The default value is norm(fc) where $f_c = [b]\{u(t)\}$ the instant load at first time step. This is used to control convergence on load.
- IterInit,IterEnd callbacks executed in non linear solver iterations. This is evaluated when entering and exiting the Newton solver. Can be used to save specific data, implement modified solvers, ...
- Jacobian string to be evaluated to generate a factored jacobian matrix in matrix or ofact object ki. Defaults are detailed for each solver, see also NLJacobianUpdate if you have the non-linear vibration tools.
- JacobianUpdate controls the update of Jacobian in Newton and quasi-Newton loops. Use 1 for updates and 0 for a fixed Jacobian (default).
- Residual Callback evaluated for residual. The default residual is method dependent.

- `InitAcceleration` optional field to be evaluated to initialize the acceleration field.
- `IterFcn` string or function handle iteration (inner loop) function. When performing the time simulation initialization, the string will be replaced by the function handle (*e.g.* `@iterNewton`). Iteration algorithms available in `fe_time` are `iterNewton` (default for basic Newton and Newmark) and `iterNewton_Sec` which implements the Newton increment control algorithm.
- `RelTol` threshold for convergence tests. The default is the OpenFEM preference `getpref('OpenFEM','THRESHOLD',1e-6);`
- `TimeVector` **optional** value of computed time steps, if exists `TimeVector` is used instead of `deltaT,Nstep`.
- `AssembleCall` **optional** callback for assembly, see `nl_spring('AssembleCall')`. When `model` and `Case` are provided as fully assembled, one can define the `AssembleCall` field as empty to tell `fe_time`not to perform any assembly. Description of assemble calls can be found in section 4.8.8 .

to control the output

- `OutInd` DOF output indices (see 2D example). This selection is based on the state DOFs which can be found using `fe_case(model,'GettDof')`.
- `OutputFcn` string to be evaluated for post-processing or time vector containing the output time steps. Examples are given below.
- `FinalCleanupFcn` string to be evaluated for final post-processing of the simulation
- `c_u, c_v, c_a` optional observation matrices for displacement, velocity and acceleration outputs. See section 4.6.4 for more details on observation matrix generation.
- `lab_u, lab_v, lab_a` optional cell array containing labels describing each output (lines of observation matrices)
- `NeedUVA [NeedU NeedV NeedA]`, if `NeedU` is equal to 1, output displacement, etc. The default is `[1 0 0]` corresponding to displacement output only.
- `OutputInit` optional string to be evaluated to initialize the output (before the time loop). The objective of this call is to preallocate matrices in the `out` structure so that data can be saved efficiently during the time integration. In particular for many time steps `out.def` may be very large and you want the integration to fail allocating memory before actually starting.

- SaveTimes optional time vector, saves time steps on disk
- Follow implements a timer allowing during simulation display of results. A basic follow mechanism is implemented (opt.Follow=1; to activate, see NLNewmark example below)). One can also define a simple waitbar with remaining time estimation, with: opt.Follow='cingui(''TimerStartWaitBar-title"Progress bar example..."'')'; More elaborate monitoring are available within the SDT optional function nl_spring (see nl_spring Follow).

Input and output options

This section details the applicable input and the output options.

Initial conditions may be provided in a model.Stack entry of type info named q0 or in an input argument q0. q0 is a data structure containing def and DOF fields as in a FEM result data structure (section 4.8). If any, the second column gives the initial velocity. If q0 is empty, zero initial conditions are taken. In this example, a first simulation is used to determine the initial conditions of the final simulation.

```
model=fe_time('demo bar');
TimeOpt=fe_time('TimeOpt Newmark .25 .5 0 1e-4 100');
TimeOpt.NeedUVA=[1 1 0];
% first computation to determine inititial conditions
def=fe_time(TimeOpt,model);

% no input force
model=fe_case(model,'remove','Point load 1');

% Setting initial conditions
q0=struct('def',[def.def(:,end) def.v(:,end)],'DOF',def.DOF);
model=stack_set(model,'curve','q0',q0);

def=fe_time(TimeOpt,model);
```

An alternative call is possible using input arguments

```
def=fe_time(TimeOpt,model,Case,q0)
```

In this case, it is the input argument q0 which is used instead of an eventual stack entry.

You may define the time dependence of a load using curves as illustrated in section 7.9 .

You may specify the time steps by giving the 'TimeVector'

```
TimeOpt=struct('Method','Newmark','Opt',[.25 .5 ],...
               'TimeVector',linspace(0,100e-4,101));
```

This is useful if you want to use non constant time steps. There is no current implementation for self adaptive time steps.

To illustrate the output options, we use the example of a 2D propagation. Note that this example also features a time dependent DofLoad excitation (see fe_case) defined by a curve, (see fe_curve), here named Point load 1.

```
model=fe_time('demo 2d');
TimeOpt=fe_time('TimeOpt Newmark .25 .5 0 1e-4 50');
```

You may specify specific output by selecting DOF indices as below

```
i1=fe_case(model,'GettDof'); i2=feutil('findnode y==0',model)+.02;
TimeOpt.OutInd=fe_c(i1,i2,'ind');
model=stack_set(model,'info','TimeOpt',TimeOpt);
def=fe_time(model); % Don't animate this (only bottom line)
```

You may select specific output time step using TimeOpt.OutputFcn as a vector

```
TimeOpt.OutputFcn=[11e-4 12e-4];
TimeOpt=feutil('rmfield',TimeOpt','OutInd');
model=stack_set(model,'info','TimeOpt',TimeOpt);
def=fe_time(model); % only two time steps saved
```

or as a string to evaluate. In this case it is useful to know the names of a few local variables in the fe_time function.

- out the structure preallocated for output.
- j1 index of the current step with initial conditions stored in the first column of out.def so store the current time step in out.def(:,j1+1).
- u displacement field, v velocity field, a acceleration field.

In this example the default output function (for TimeOpt.NeedUVA=[1 1 1]) is used but specified for illustration

```
TimeOpt.OutputFcn=['out.def(:,j1+1)=u;' ...
                   'out.v(:,j1+1)=v;out.a(:,j1+1)=a;'];
model=stack_set(model,'info','TimeOpt',TimeOpt);
def=fe_time(model); % full deformation saved
```

This example illustrates how to display the result (see feplot) and make a movie

```
cf=feplot(model,def);
fecom('ColorDataEvalA');
fecom(cf,'SetProp sel(1).fsProp','FaceAlpha',1,'EdgeAlpha',0.1);
cf.ua.clim=[0 2e-6];fecom(';view2;AnimTime;ch20;scd1e-2;');
st=fullfile(getpref('SDT','tempdir'),'test.gif');
fecom(['animMovie ' st]);fprintf('\nGenerated movie %s\n',st);
```

Note that you must choose the Anim:Time option in the feplot GUI.

You may want to select outputs using observations matrix

```
model=fe_time('demo bar'); Case=fe_case('gett',model);
i1=feutil('findnode x>30',model);
TimeOpt=fe_time('TimeOpt Newmark .25 .5 0 1e-4 100');
TimeOpt.c_u=fe_c(Case.DOF,i1+.01);          % observation matrix
TimeOpt.lab_u=fe_c(Case.DOF,i1+.01,'dofs'); % labels

def=fe_time(TimeOpt,model);
```

If you want to specialize the output time and function you can specify the SaveTimes as a time vector indicating at which time the SaveFcn string will be evaluated. A typical TimeOpt would contain

```
TimeOpt.SaveTimes=[0:Ts:TotalTime];
TimeOpt.SaveFcn='My_function(''Output'',u,v,a,opt,out,j1,t);';
```

Cleanup

The field FinalCleanupFcn of the TimeOpt can be used to specify what is done just after the time integration.
fe_simul provides a generic clean up function which can be called using
opt.FinalCleanupFcn='fe_simul(''fe_timeCleanup'',model)';
If the output has been directly saved or from iiplot it is possible to load the results with the same display options than for the fe_timeCleanup using fe_simul('fe_timeLoad',fname)';

Some command options can be used:

- -cf *i* stores the result of time integration in the stack of iiplot or feplot figure number *i*. i=-1 can be specified to use current iiplot figure and i=-2 for current feplot figure. Displacements are stored in curve,disp entry of the stack. Velocities and accelerations (if any)

are respectively stored in the curve,vel and curve,acc stack entries. If command option -reset is present, existent stack entries (disp, vel, acc, etc. ...) are lost whereas if not stack entries name are incremented (disp(1), disp(2), etc. ...).

- '-ExitFcn"*AnotherCleanUpFcn*"' can be used to call an other clean up function just after fe_simul('fe_timeCleanUp') is performed.
- -fullDOF performs a restitution of the output on the unconstrained DOF of the model used by fe_time.

  -restitFeplot adds a .TR field to the output to allow deformation on the fly restitution in feplot. These two options cannot be specified simultaneously.
- Command option -rethrow allows outputting the cross reference output data from iiplotor feplotif the option -cf-1 or -cf-2 is used.

newmark

For the Newmark scheme, TimeOpt has the form

```
TimeOpt=struct('Method','Newmark','Opt',Opt)
```

where TimeOpt.Opt is defined by

```
[beta gamma t0 deltaT Nstep]
```

beta and gamma are the standard Newmark parameters [37] ([0 0.5] for explicit and default at [.25 .5] for implicit), t0 the initial time, deltaT the fixed time step, Nstep the number of steps.

The default residual is `r = (ft(j1,:)*fc'-v'*c-u'*k)';` (notice the sign change when compared to NLNewmark).

This is a simple 1D example plotting the propagation of the velocity field using a Newmark implicit algorithm. Rayleigh damping is declared using the info,Rayleigh case entry.

```
model=fe_time('demo bar');
data=struct('DOF',2.01,'def',1e6,...
            'curve',fe_curve('test ricker dt=1e-3 A=1'));
model = fe_case(model,'DOFLoad','Point load 1',data);
TimeOpt=struct('Method','Newmark','Opt',[.25 .5 3e-4 1e-4 100],...
               'NeedUVA',[1 1 0]);
def=fe_time(TimeOpt,model);
```

```
% plotting velocity (propagation of the signal)
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
```

dg

The time discontinuous Galerkin is a very accurate time solver [51] [52] but it is much more time consuming than the Newmark schemes. No damping and no non linearities are supported for Discontinuous Galerkin method.

The options are `[unused unused t0 deltaT Nstep Nf]`, `deltaT` is the fixed time step, `Nstep` the number of steps and `Nf` the optional number of time step of the input force.

This is the same 1D example but using the Discontinuous Galerkin method:

```
model=fe_time('demo bar');
TimeOpt=fe_time('TimeOpt DG Inf Inf 0. 1e-4 100');
TimeOpt.NeedUVA=[1 1 0];
def=fe_time(TimeOpt,model);

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

For the non linear Newmark scheme, `TimeOpt` has the same form as for the linear scheme (method `Newmark`). Additional fields can be specified in the `TimeOpt` data structure

Jacobian	string to be evaluated to generate a factored jacobian matrix in matrix or ofact object ki. The default jacobian matrix is 'ki=ofact(model.K{3}+2/dt*model.K{2}' +4/(dt*dt)*model.K{1});'
Residual	Defines the residual used for the Newton iterations of each type step. It is typically a call to an external function. The default residual is 'r = model.K{1}*a+model.K{2}*v+model.K{3}*u-fc;' where fc is the current external load, obtained using (ft(j1,:)*fc')' at each time step.
IterInit	evaluated when entering the correction iterations. This can be used to initialize tolerances, change mode in a co-simulation scheme, etc.
IterEnd	evaluated when exiting the correction iterations. This can be used to save specific data, ...
IterFcn	Correction iteration algorithm function, available are iterNewton (default when omitted) or iterNewton_Sec. Details of the implementation are given in the staticNewton below.
MaxIterSec	for iterNewton_Sec applications (see staticNewton).
ResSec	for iterNewton_Sec applications (see staticNewton).

Following example is a simple beam, clamped at one end, connected by a linear spring at other end and also by a non linear cubic spring. The NL cubic spring is modeled by a load added in the residual expression.

```
% Get simple test case for NL simulation in sdtweb demosdt('BeamEndSpring')
model=demosdt('BeamEndSpring'); % simple example building
opt=stack_get(model,'info','TimeOpt','GetData');
disp(opt.Residual)
opt.Follow=1; % activate simple monitoring of the
%               number of Newton iterations at each time step
def=fe_time(opt,model);
```

staticNewton

For non linear static problems, the Newton solver iterNewton is used. TimeOpt has a similar form as with the NLNewmark method but no parameter Opt is used.

An increment control algorithm iterNewton_Sec can be used when convergence is difficult or slow (as it happens for systems showing high stiffness variations). The Newton increment Δq is then the first step of a line search algorithm to optimize the corrective displacement increment $\rho\Delta q, \rho \in \mathbf{R}$ in the iteration. This optimum is found using the secant iteration method. Only a few optimization iterations are needed since this does not control the mechanical equilibrium but only the relevance of the Newton increment. Each secant iteration requires two residual computations, which can be

costly, but more efficient when a large number of standard iterations (matrix inversion) is required to obtain convergence.

Fields can be specified in the TimeOpt data structure

Jacobian	defaults to 'ki=ofact(model.K{3});'
Residual	defaults to 'r = model.K{3}*u-fc;'
IterInit IterEnd	and IterEnd are supported see fe_time TimeOpt
MaxIterSec	Maximum secant iterations for the iterNewton_Sec iteration algorithm. The default is 3 when omitted.
ResSec	Residual evaluation for the secant iterations of the iterNewton_Sec iteration algorithm. When omitted, fe_timetries to interpret the Residual field. The function must fill in the secant residual evaluation r1 which two columns will contain the residual for solution rho(1)*dq and rho(2)*dq. The default ResSec field will be then 'r1(:,1) = model.K{3}*(u-rho(1)*dq)-fc; r1(:,2) = model.K{3}*(u-rho(2)*dq)-fc;'.

Below is a demonstration non-linear large transform statics.

```
%  Sample mesh, see script with sdtweb demosdt('LargeTransform')
model=demosdt('largeTransform'); %

% Now perform the Newton loop
model=stack_set(model,'info','TimeOpt', ...
   struct('Opt',[],'Method','StaticNewton',...
   'Jacobian','ki=basic_jacobian(model,ki,0.,0.,opt.Opt);',...
   'NoT',1, ... % Don't eliminate constraints in model.K
   'AssembleCall','assemble -fetimeNoT -cfield1', ...
   'IterInit','opt=fe_simul(''IterInitNLStatic'',model,Case,opt);'));
model=fe_case(model,'setcurve','PointLoad', ...
    fe_curve('testramp NStep=20 Yf=1e-6')); % 20 steps gradual load
def=fe_time(model);
cf=feplot(model,def); fecom(';ch20;scc1;colordataEvalZ'); % View shape
ci=iiplot(def);iicom('ch',{'DOF',288.03}) % View response
```

numerical damping for Newmark, HHT-alpha schemes

You may want to use numerical damping in a time integration scheme, the first possibility is to tune the Newmark parameters using a coefficient α such that $\beta = \frac{(1+\alpha)^2}{4}$ and $\gamma = \frac{1}{2} + \alpha$. This is known

to implement too much damping at low frequencies and is very depending on the time step [37].

A better way to implement numerical damping is to use the HHT-α method which applies the Newmark time integration scheme to a modified residual balancing the forces with the previous time step.

For the HHT-α scheme, `TimeOpt` has the form

```
TimeOpt=struct('Method','nlnewmark','Opt',Opt,...
               'HHTalpha',alpha)
```

where `TimeOpt.Opt` is defined by

```
[beta gamma t0 deltaT Nstep]
```

`beta` and `gamma` are the standard Newmark parameters [37] with numerical damping, `t0` the initial time, `deltaT` the fixed time step, `Nstep` the number of steps.

The automatic TimeOpt generation call takes the form `[alpha unused t0 deltaT Nstep]` and will compute the corresponding β, γ parameters.

This is a simple 1D example plotting the propagation of the velocity field using the HHT-α implicit algorithm:

```
model=fe_time('demo bar');
TimeOpt=fe_time('TimeOpt hht .05 Inf 3e-4 1e-4 100');
TimeOpt.NeedUVA=[1 1 0];
def=fe_time(TimeOpt,model);
```

The call

```
TimeOpt=fe_time('TimeOpt hht .05 Inf 3e-4 1e-4 100');
```

is strictly equivalent to

```
TimeOpt=struct('Method','nlnewmark',...
               'Opt',[.275625 .55 3e-4 1e-4 100],...
               'HHTalpha',.05);
```

Theta

The θ-method is a velocity based solver, whose formulation is given for example in [53, 54]. It considers the acceleration as a distribution, thus relaxing discontinuity problems in non-smooth dynamics. Only a linear implementation is provided in `fe_time`. The user is nevertheless free to implement a non-linear iteration, through his own `IterFcn`.

This method takes only one integration parameter for its scheme, θ set by default at 0.5. Any values between 0.5 and 1 can be used, but numerical damping occurs for $\theta > 0.5$.

The `TimeOpt.Opt` takes the form `[theta unused t0 deltaT Nstep]`.

This is a simple 1D example plotting the propagation of the velocity field using the θ-Method:

```
model=fe_time('demo bar');
TimeOpt=fe_time('TimeOpt theta .5 0 3e-4 100');
def=fe_time(TimeOpt,model);
```

Euler

This method can be used to integrate first order problem of the form $M\dot{q} + Kq = F$. One can use it to solve transient heat diffusion equation (see `p_heat`).

Integration scheme is of the form $q_{n+1} = q_n + (1-\theta)h\dot{q}_n + \theta h\dot{q}_{n+1}$
θ can be define in `opt.Opt(1)`. Explicit Euler ($\theta = 0$) is not implemented at this time. Best accuracy is obtained with $\theta = \frac{1}{2}$ (Crank-Nicolson).

See also

`fe_mk`, `fe_load`, `fe_case`

of_time

Purpose

The of_time function is a low level function dealing with CPU and/or memory consuming steps of a time integration.

The **case sensitive** commands are

lininterp	linear interpolation.
storelaststep	pre-allocated saving of a time step in a structure with fields initially built with struct('uva',[u,v,a],'FNL',model.FNL)
interp	Time scheme interpolations (low level call).
-1	In place memory assignment.

lininterp

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 abscissa 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)',zeros(1,3))
```

Warning : this command modifies the variable last within a given function this may modify other identical constants in the same m-file. To avoid any problems, this variable should be generated using zeros (the Matlab function) to assure its memory allocation independence.

The storelaststep command makes a deep copy of the displacement, velocity and acceleration fields (stored in each column of the variable uva.uva in the preallocated variables u, v and a following the syntax:

```
of_time('storelaststep',uva,u,v,a);
```

interp

This command performs transient numerical scheme response interpolations. It is used by fe_time when the user gives a TimeVector in the command. In such case the output instants do not corre-

spond to the solver computation instants, the approached output instants must thus be interpolated from the solver instants using the numerical scheme quadrature rules.

This command uses current solver instant `t1` and the last instant step `t0` of the solver `uva`. The `uva` matrix is stored in `Case` and contains in each column, displacement, velocity and acceleration at `t0`. The interpolation strategy that is different for each numerical scheme depends on the arguments given to `of_time`.

Warning : this command modifies `out.def` at very low level, `out.def` thus cannot be initialized by simple numerical values, but by a non trivial command (use `zeros(1)` instead of `0` for example) to ensure the unicity of this data in memory.

For a `Newmark` or `HHT-alpha` scheme, the low level call command is

```
of_time ('interp', out, beta,gamma,uva,a, t0,t1,model.FNL);
```

where `beta` and `gamma` are the coefficients of the Newmark scheme, first two values of `opt.Opt`.

Thus the displacement (u_1) and velocity (v_1) at time `t1` will be computed from the displacement (u_0), velocity (v_0), acceleration (a_0) stored in `uva`, the new acceleration `a` (a_1), and the time step ($h = t1 - t0$) as

$$\begin{cases} v_1 = v_0 + h(1-\gamma)a_0 + h\gamma a_1 \\ u_1 = u_0 + hv_0 + h^2(\frac{1}{2} - \beta)a_0 + h^2\beta a_1 \end{cases} \tag{10.23}$$

NL force (model.FNL) is linearly interpolated.

For the `Theta-Method` scheme, the low level command is

```
of_time ('interp', out, opt.Opt(1),[],uva,v, t0,t1,model.FNL);
```

Thus the displacement (u_1) at time `t1` will be computed from the displacement (u_0), velocity (v_0), stored in `uva`, the new velocity `v` (v_1), and the time step ($h = t1 - t0$) as

$$u_1 = u_0 + h(1-\theta)v_0 + h\theta v_1 \tag{10.24}$$

For the `staticnewton` method, it is possible to use the same storage strategy (since it is optimized for performance), using

```
of_time ('interp', out, [],[], [],u, t0,t1,model.FNL);
```

In this case no interpolation is performed.

Please note that this low-level call uses the internal variables of fe_time at the state where is is evaluated. It is then useful to know that inside fe_time:

- current instant computed is time tc=t(j1+1) using time step dt, values are t0=tc-dt and t1=tc.
- uva is generally stored in Case.uva.
- the current acceleration, velocity or displacement values when interpolation is performed are always a, v, and u.
- The out data structure must be preallocated and is modified by low level C calls. Expected fields are

def	displacement output, must be preallocated with size length(OutInd) x length(data)
v	velocity output, must be preallocated with size length(OutInd) x length(data)
a	acceleration output (when computed) must be preallocated with size length(OutInd) x length(data)
data	column vector of output times
OutInd	int32 vector of output indices, must be given
cur	[Step dt], must be given
FNL	possibly preallocated data structure to store non-linear loads. FNL.def must be length(model.FNL) by size(out.data,1) (or possibly size(out.FNL.data,1), in this case fieldnames must be def,DOF,data,cur)

- non linear loads in model.FNL are never interpolated.

-1

This command performs in place memory assignment of data. It is used to avoid memory duplication between several layers of code when computation data is stored at high level. One can thus propagate data values at low level in variables shared by several layers of code without handling output and updates at each level.

The basic syntax to fill-in preallocated variable r1 with the content of r2 is i0 = of_time(-1,r1,r2);. The output i0 is the current position in r1 after filling with r2.

It is possible to use a fill-in offset i1 to start filling r1 with r2 from index position i1 : i0 = of_time([-1 i1],r1,r2);.

To avoid errors, one must ensure that the assigned variable is larger than the variable to transmit. The following example illustrates the use of this command.

```
% In place memory assignment in vectors with of_time -1
r1=zeros(10,1); % sample shared variable
r2=rand(3,1); % sample data
% fill in start of r1 with r2 data
of_time(-1,r1,r2);
% fill in start of r1 with r2 data and
% get current position in r1
i0=of_time(-1,r1,r2);
% i0 is current pos
% fill in r1 with r2+1
% with a position offset
i0=of_time([-1 i0],r1,r2+1);
```

See also

fe_time

idcom

Purpose

UI command functions for standard operations in identification.

Syntax

```
idcom('CommandString');
```

Description

The idcom command should only be used for script purpose. Most commands correspond to the underlying button callbacks of the Ident table (see section 8.2.6). Chapter 2 presents the interactive way to perform a modal identification with SDT using of the dedicated dock Id.

idcom provides a simple access to standard operations in identification. The way they should be sequenced is detailed in section 2.5 which also illustrates the use of the associated GUI.

idcom is always associated with an iiplot figure. Information on how to modify standard plots is given under iicom. The datasets used by idcom are described in section 2.5 . Methods to access the data from the command line are described in section 2.1.2 . Identification options stored in the figure are detailed under the idopt function.

idcom(ci) turns the environment on, idcom(ci,'Off') removes options but not datasets.

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 opens or refreshes the current idcom figure.

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 ci.Stack{'test'}.

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 stored in ci.Stack{'IdAlt'} with a pole .po and residue .res field. FRFs are resynthesized into ci.Stack{'IdFrf'} (which is overlaid to ci.Stack{'Test'} in iiplot). If, based on the plot(s), the estimate seems good it should be added to the current pole set ci.Stack{'IdMain'} using ea.

ea

Add alternate poles to the main set. If appropriate modes are present in ci.Stack{'IdAlt'} (after using the e or f commands for example), they should be added to the main pole set ci.Stack{'IdMain'} using the ea command. These poles can then be used to identify a multiple pole broadband model with idcom est and idcom eup commands.

If all poles in ci.Stack{'IdAlt'} are already in ci.Stack{'IdMain'}, 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 ci.Stack{'IdMain'}. 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 ci.Stack{'IdAlt'} so that an ea command will undo the removal.

est[,local,localpole]

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 ci.Stack{'IdMain'} but does not update it. The results are a residue matrix ci.Stack{'IdMain'}.res, and corresponding FRF ci.Stack{'IdFrf'} (which is overlaid to ci.Stack{'Test'} in iiplot). In most cases the estimate can be improved by optimizing the poles using the eup or eopt commands.

estLocal only estimates residues of poles in the range selected by ci.IDopt. You perform a series of local estimates over selected bands by providing these bands or using narrow band around each pole with estLocalPole.

gartid

```
idcom('w0');idcom est
def_global=ci.Stack{'IdMain'}; % broadband estimate
```

```
idcom('estlocal',{[6 7],[15 17],[31 38],[48 65]});
def_local=ci.Stack{'IdMain'}; % estimate by multiple local bands
```

eup *dstep fstep* [local, num *i* , iter *j*]

Update of poles. eup uses id_rc to update the poles of a multiple pole model based data within ci.IDopt.SelectedRange. This update is done through a non-linear optimization of the pole locations detailed in section 2.6.5 . The results are updated modes ci.Stack{'IdMain'} (the initial ones are stored in ci.Stack{'IdAlt'}), and corresponding FRF ci.Stack{'IdFrf'} (which is overlaid in iiplot).

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 option 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 option num *i* where *i* gives the indices in IdMain 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 ci.Stack{'IdMain'}.po are all the information needed to obtain the full model estimate. You should save this information in a text file (use idcom('TableIdMain') to generate a clean output) 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 2.2.2).

`eopt [local, num i, seq]`

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 (3.3) so that using it rarely hurts.

`eoptSeq` seeks to optimize all poles of the band. This is commonly efficient when starting from stabilization results.

`find`

Find a pole. This command detects minima of the MMIF that are away from poles of the current model `ci.Stack{'IdMain'}.po` 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 (`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 `ci.Stack{'IdAlt'}.po` 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 `Test`. `nm` must be less than the number of sensors. The results are transformed to the residue form which gives poles and residues in `IdMain`, and corresponding FRF `IdFrf` (which is overlaid to `Test` in `iiplot`.

`mass i`

Computes the generalized mass at address `i`. If the identified model contains complex residues (`ci.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

$$c_{Col}\phi_j = \phi_j^T b_{Col} = \sqrt{R_{jCol}} = (m_{jCol})^{-1/2} \tag{10.25}$$

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 `Test` with numerators of degree *nn* and denominators of degree *nd*. The corresponding FRFs are stored in `IdFrf` (which is overlaid to `Test` in `iiplot`).

`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 2.2

idopt

Purpose

handling of options used by the identification related routines.

Description

idopt is the function handling identification options. Identification options associated with idcom figures are used when generating new identifications. They should be modified using the ci.IDopt pointer or the IDopt tab in the figure. In the text output below

```
>> ci=idcom; ci.IDopt
  (ID options in figure(2)) =
   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 ]
```

currently selected value are shown between braces { } and alternatives are shown.

After performing an identification, the options used at the time are copied to the result. Thus the ci.Stack{'IdMain'}.idopt is a copy of the figure options when the identification was performed. Some manipulations possible with the res2nor,res2ss,id_nor, ... functions may require modifications of these options (which are different from the idcom figure options.

The *SDT* handle object used to store options is very permissive in the way to change values from the command line (for GUI operation use the IDopt tab). ci.IDopt.OptName=OptValue sets the option. OptName need only specify enough characters to allow a unique option match. Thus ci.IDopt.res and ci.IDopt.ResidualTerms are equivalent. Here are a few examples

```
demosdt('demoGartIdEst');ci=idcom;
ci.IDopt.Residual=0; % modify estimation default
ci.IDopt.Selected=[100  2000];
ci.IDopt.Po='Hz';
ci.IDopt % changed
ci.Stack{'IdMain'}.idopt % not changed until new identification
```

The following is a list of possible options with indications as to where they are stored. Thus

ci.IDopt.res=2 is simply a user friendly form for the old call ci.IDopt(6)=2 which you can still use.

Res		Residual terms selection (stored in ci.IDopt(1)) and corresponding to (5.26)
	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 ci.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 ci.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 ci.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 ci.IDopt(4:5))
Fitting		model (see res page 212, stored in ci.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 ci.IDopt(7:8))
Recip		method selection for the treatment of reciprocity (stored in ci.IDopt(12))
	1	means that only iC1 (ci.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 214)

To make a copy of the data, and no longer point to the figure, use ci.IDopt.GetData.

`iop2 = idopt` returns a *SDT* `handle` to a set options that may differ from those of used by `idcom`.

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 [55] 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

$$\left\{ \begin{array}{c} \dot{p} \\ \ddot{p} \end{array} \right\} = \left[\begin{array}{cc} 0 & I \\ -K_T & -C_T \end{array} \right] \left\{ \begin{array}{c} p \\ \dot{p} \end{array} \right\} + \left[\begin{array}{c} 0 \\ b_T \end{array} \right] \{u\} \quad \text{and} \quad \{y\} = \left[\begin{array}{cc} c_T & 0 \end{array} \right] \left\{ \begin{array}{c} p \\ \dot{p} \end{array} \right\} \tag{10.26}$$

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 (`ci.IDopt.ns`).

```
ci=iicom('curveload sdt_id');
R1=ci.Stack{'Test'};
[a,b,c,d] = id_dspi(R1.xf,ones(size(R1.w)),R1.w,R1.idopt,4);
```

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 `ci.IDopt.SelectedRange`. `y` is supposed to be a displacement. If `ci.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.

```
NOR                = id_nor(ci.Stack{'IdMain'})
NOR                = id_nor( ... )
[om,ga,phib,cphi] = id_nor( ... )
[new_res,new_po]   = id_nor( ... )
[ ... ]            = id_nor(IdResult,ind,opt,res_now)
```

Description

`id_nor` is meant to provide an optimal transformation (see details in [12] or section 2.8.3) between the residue (result of `id_rc`) and non-proportionally damped normal mode forms

$$\{y(s)\} = \sum_{j=1}^{2N} \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j}\{u\} \quad \text{and} \quad \begin{array}{c}\left[Is^2+\Gamma s+\Omega^2\right]\{p\} = \left[\phi^T b\right]\{u\} \\ \{y\} = [c\phi]\{p\}\end{array} \tag{10.27}$$

The output arguments are either

- the standard normal mode model `freq,ga,phib,cphi` (see `nor`) 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 212) 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

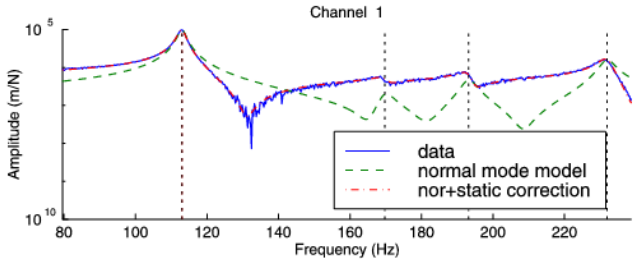
```
[new_res,po] = id_nor(res,po,idopt,ind,opt,old_res)
```

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

```
ci=demosdt('demo gartidest')
if ci.Stack{'Test'}.dof(4,2)~=1012.03;% Needed to have positive driving point FRFs
 ci.Stack{'Test'}.xf=-ci.Stack{'Test'}.xf;
 ci.Stack{'Test'}.dof(:,2)=1012.03; idcom('est');
end
nor = id_nor(ci.Stack{'IdMain'});
ci.Stack{'curve','IIxh'}=nor2xf(nor,ci.Stack{'Test'}.w,'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. [8] 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.

```
[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 2.2 .

For a given set of poles, idrc(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 and corresponds to the theory in section 2.6.5 .

As detailed in section 2.6 , the poles can (and should) be tuned [9] 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, see ii_pof for the format). 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. To save the poles to a text file, use idcom Table. 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 5.6). 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 2.2
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.

```
OUT = id_rm(IN,multi)
[psib,cpsi,new_res,new_po] = id_rm(res ,po,ci.IDopt)
[phib,cphi,new_res,new_po] = id_rm(Rres,po,ci.IDopt)
[psib,cpsi,new_res,new_po] = id_rm(res ,po,ci.IDopt,multi)
OUT = id_rm('Command',Curve) % See accepted commands at end of doc
```

Description

id_rm is more easily called using the idcom GUI figure Postprocessing tab, see section 2.8 .

IN is a data structure (see 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 ci.IDopt as arguments are obsolete and only maintained for backward compatibility reasons.

As shown in more detail in section 2.8 , 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

$$[\alpha(s)] = \sum_{j=1}^{2N} \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j} \approx \sum_{j=1}^{2N} \frac{[R_j]}{s-\lambda_j} \tag{10.28}$$

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 2.8.1). 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.

PermuteIO

The `C1=id_rm('permuteIO',C1);` command renumbers transfer functions to use the reference order of sensors at each actuator in the case of hammer tests where there are more input locations than outputs.

FixSign

The `C1=id_rm('FixSign',C1);` applies sign changes on sensors and inputs to generate positive sign transfers or modeshapes.

Mass

`id_rm('Mass',Id);` is the low level implementation of generalized mass extraction.

See also

idcom, id_rc, id_nor, the demo_id demonstration

iicom

Purpose

UI command function for FRF data visualization.

Syntax

```
iicom CommandString
iicom(ci,'CommandString') % specify target figure with pointer
out = iicom('CommandString')
```

Description

`iicom` is a standard *UI command* function which performs operations linked to the data visualization within the `iiplot` interface. A tutorial can be found in section 2.1 .

Commands are text strings telling `iicom` what to do. If many `iiplot` figures are open, one can define the target giving an `iiplot` figure handle `ci` as a first argument.

`iicom` uses data stored in a stack (see section 2.1.2). `iicom` does not modify data. 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;`

The `commode` help details generic command building mechanisms. 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.

`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 `iicom Sub` command). The button indicates the number of the current axis. Pressing the button executes the `ca+` command.

ch+, ch-, ch[+,-]*i* : next/previous

Next/Previous − +. These commands/buttons are used to scan through plots of the same kind. For iiplot axes, this is applied to the current data sets. 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*, chall, ... select channel

Display channels/poles/deformations ***i***. *Channels* refer to columns of datasets, poles or deformations. 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.

For curve Multi-dim curve with dimension labels in the .Xlab field,ChAllMyLabel selects all channels associated with dimension MyLabel. This can be used to show responses at multiple operating conditions (typically stored as third or fourth dimension of curve.Y).

For multi-channel curves one can define the dimension name referring to the Xlab field in a cell array iicom(ci,'ch','Xlabname',i). For this to work properly note that all Xlabname entries must be different (*e.g.* several Unknown entries must thus be avoided).

```
% Build a multi-dim curve, see sdtweb('demosdt.m#DemoGartteCurve')
r1=demosdt('demoGartteCurve')
ci=iicom('curveInit','Example',r1);
iicom('ChAllzeta') % All channels that correspond to 'zeta' r1.Xlab{4}
% Cell selection with Xlab string and indices (each row picks a dimension)
iicom('ch',{'Output DOFs',4;'Input DOFs',[1,2]}) % Accessible with 'pick' button
iicom('curtabChannel')
```

Cursor, ods

The cursor is usually started with the axes context menu (right click on a given axis).

iicom CursorOnFeplotshows a cursor on the iiplot curve that let you show corresponding time deformation in feplot.

fecom CursorNodeIiplot gives more details.

iicom('ods') provides an *operational deflection shape* cursor.

Curve [Init,Load,Save,Reset, ...]

These commands are used to manipulate datasets.
Most of them are of the form `iicom('Curve...',CurveNames)`. Then `CurveNames` can be a string with a curve name, a cell array of string with curve names or a regular expression (beginning by `#`) to select some curve names. If `CurveNames` is omitted, a curve a dialog box is opened to select targeted curves. Otherwise these commands can be accessed through the GUI, in the `Stack` tab of the `iiplot` properties figure.

- `CurveInit` is used to initialize a display with a new dataset. `iicom('CurveInit','Name',C1)` is used to initialize a display with a new dataset. `iicom('CurveInit','Name',C1)` adds a `'curve','Name'` entry and displays this set in a new tab. To add dans display multiple curves use

  ```
  iicom('CurveInit',{'curve','N1',C1; 'curve','N2',C2})
  ```

  The field `PlotInfo` can be used to control how this initial display is performed.

- `CurveLoad` lets you load datasets.
  `iicom('CurveLoad` *`FileName`*`')` loads curves stored in *`Filename`*.
  `iicom('CurveLoad')` opens a dialog box to choose the file containing curves to load. If the file contains multiple curves, one can select the curves to be loaded in a cell array given as a second argument. For example,

  ```
  ci=iicom('CurveLoad','gartid.mat')
  ```

  loads the `gartid` data in an `iiplot` figure. Command option `-append` (`iicom(ci,'CurveLoad -append MyFile')`) lets you append loaded curves to existing curves in the stack (by default existing curves are replaced). Command option `-hdf` (`iicom(ci,'CurveLoad -hdf MyFile')`) lets you load curves under the `sdthdf`format. Only pointers to the data stacked in `iiplot`are thus loaded. Visualizations and data transformation can be performed afterwards. Command option `-back` does not generate any visualization in `iiplot`. This can be useful in combination to `-hdf`, as the user can then fully control the data loaded in RAM.

- `CurveSave` lets you save `iiplot` stack data.
  `iicom('CurveSave` *`FileName`*`',CurveNames)` saves the curves `CurveNames` in the `.mat` file given by *`FileName`*. If *`FileName`* is omitted a GUI is opened. To save more than 2 GB of data, or to save in the new MATLAB file formats (`-v7.3`), use the SDT `V6Flag`:
  `setpref('SDT','V6Flag','-v7.3')`.

  ```
  fname=fullfile(sdtdef('tempdir'),'IicomSaveTestmat')
  iicom(['CurveSave' fname],{'IIxi';'IdMain'})
  ```

- CurveNewId *CurveName* opens new iiplot figure for identification of the curve *CurveName* of the ci stack with idcom.
  iicom('CurveLoadId',*FileName*) loads from *FileName* into for identification.

- CurveRemove removes the curves from the stack of the iiplot figure.
  iicom('CurveRemove',CurveNames);

- CurveReset defines an empty curve stack to renew your work.

- CurveJoin combines datasets that have comparable dimensions. In particular first dimension (time, frequencies ...) must be the same. For example it is useful to combine dataset from parameter studies (same dimension). iicom('CurveJoin',CurveNames);
  Curves targeted by CurveNames (or selected curves in iiplot) are joined and replace the first curve in the iiplot stack.

- CurveCat concatenates dataset that have the same dimensions. For example it is useful to combine dataset from successive time simulation. Syntax is the same as for iicom CurveJoin command. One can use following command options:
  - -follow to remove last value of first abscissa before concatenate.
  - -shift to shift abscissa of second dataset of the last value of first dataset abscissa.

Dock Id, MAC, TestBas

Starting with SDT 7, classical SDT uses are guided through multiple figures combined in docks.

- DockId is used for identification of modeshapes.

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 feplot or iiplot figure, you will prefer the calling format cf=iiplot; set(cf.ga(1:2),'xgrid','on').

`head [Main,Text,Clear]`

Note : the preferred approach is now to define fixed displays using `comgui objSet` commands stored in the curve `PlotInfo` `ua.axProp` entry. For example

```
C1=fe_curve('testSin T 0.2',linspace(0,10,100e3));
C1.Xlab={'Time','Resp'};
r1={'@title',{'String','Main Title','FontSize',16}};
C1=sdsetprop(C1,'PlotInfo.ua.axProp',r1{:});
iicom('curveinit','SineWithFixedTitle',C1);
```

For backward compatibility, header axes are still supported (the change is to `objSet` allows better tab switching). Header axes are common to all plot functions and span the full figure area (normalized position `[0 0 1 1]`). You can get a pointer to this axis with `cf.head` and add any relevant object there.

```
ci=iicom('curveload','gartid'); % Load a test case
h=text(0,0,'Main Title', ...
 'parent',ci.head,'unit','normalized','position',[.05 .95], ...
 'fontsize',20,'fontname','Times', ...
 'tag','iimain');
iimouse('textmenu',h); % Allow Editing
```

`iicom('HeadClear')` deletes all objects from the header axis of the current figure.

`IIx`*`Data set selection`* `iicomIIx:`*`name`* `[On,Off,Only], cIIx ...`

Curve set selection for display in the current axis.

`IIx:TestOnly` displays the `ci.Stack{'Test'}` data set only in all axes (`on` and `off` turn the display on or off respectively). By adding a `c` in front of the command (`cIIx:Test` 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` menu (applies to all axes) or axis context menu applies to (current axis).

The alternate calling format `iicom('iix',{'Test','IdFrf'})` can be used to specify multiple sets to display. `iicom('iixOnly',{'Test','IdFrf'})` will display those two sets only.

`IIxf, IIxe, IIxh, IIxi [On,Off]` are still supported for backward compatibility.

`Polar`

Polar plots are used for cases where the abscissa is not the standard value. Accepted values (use a command of the form `Polar` *`val`*) are

- -1 abscissa is the channel before the one displayed. In a curve with channels [X Y] display Y, channel 2, and use X,channel 1, as abscissa.
- xi uses i^{th} column of def.data when displaying FEM time signals. This is typically used when this second column is an other form of abscissa (angle for rotating machines, ...)
- i with i¿0 uses the specified channel as abscissa.
- Off or 0 turns off polar plots.

PoleLine [,c] [,3], IIpo, ...

Pole line display. are dotted vertical lines placed at relevant abscissa values. These lines can come from

- standard curves with an curve.ID field, see ii_plp Call from iiplot.
- frequencies of poles in ci.Stack{'IdMain'} in black and ci.Stack{'IdAlt'} in red.

By itself, PoleLine toggles the state of pole line display. The c option 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).

Low level commands IIpo and IIpo1 are low level commands force/disable display of pole lines in the main identified model
ci.Stack{'IdMain'}.po or the alternate set ci.Stack{'IdAlt'}.po. With cIIpo the choice is only applied to the current axis. These options are usually accessed through menus.

ImWrite, ...

comgui ImWrite is the generic command used to generate a clean printout of figures. It supports many basic capabilities, filename generation, cropping, ... When using iiplot and feplot, it may often be interesting to generate **multiple images** by scanning through a selected range of channels. A command of the form iicom(cf,'ImWrite',RO) is then used with RO a structure containing generic image capture fields (see comgui ImWrite) and fields specific to multi-image capture

- .ShowFcn the callback that is executed for each image to be generated. The default is fecom(cf,sprintf('ch %i',ch)); for feplot. The loop index is j1.

- .ch a vector of channel indices that will give an index for each image. With the string all, all the channels are used.
- .ImWrite is the command used to call comgui with the default 'imwrite -ftitle'.
- .FileName if present replaces any other file name generation mechanism. Your ShowFcn callback can thus implement your own file name generation mechanism.
- .Movie can be a structure for movie generation using fecom AnimMovie.
- .HtmWidth can specify an HTML view size which differs from the image size. The input is either a string in the format width=val height=val1, or a line with 4 columns in the format [Width Height MaxWidth MaxHeight], it is possible to let free a value by provided Inf instead of a numerical value. At least Height *or* Width must be defined. Depending on the input, the behavior is
  - if a scalar is given or if the Height is set to Inf,the width is fixed and the height is set to keep the image ratio. If a MaxHeight is provided and the resulting height overcomes it, the width is adapted to maximize the possible size.
  - if Width is set to Inf, the height must be defined and the width is set to keep the image ratio. If a MaxWidth is provided and the resulting width overcomes it, the height is adapted to maximize the possible size.
  - is both Width and Height are provided, the values are fixed and non further control is performed.
- .RestoreFig=1 can be used to restore the figure and display after image generation.
- .RelPath optional integer giving the level of relative path to be retained (1 keeps just the file name, 2 the directory containing the images, ...). This is useful to create HTML report files that can be moved.

To automate figure generation, it is typically desirable to store image capture information in the set of deformations or the curve. A curve.ImWrite field in iiplot can be used to predefine the option structure, for user defined dynamic change of settings, defining a ua.PostFcn callback (see iiplot PlotInfo) is typically the appropriate approach. For feplot, def.ImWrite is used for multi-image capture but more evolved file name generation is found using comgui def.Legend.

```
% Example of 4 views in feplot
 cf=demosdt('DemoGartFEplot')
 cingui('PlotWd',cf,'@OsDic(SDT Root)','FniiLeg');
 cf.def=sdsetprop(cf.def,'Legend', ...
     'string',{'Garteur FE';'$Title'}) % Define a two line title
```

```
RO=comgui('imfeplot4view'); % Predefined strategy to generate 4 views
comgui('PlotWd',cf,'FileName', ...
 {'@PlotWd','Root','@ii_legend(1:2)','@cf.ga.View','.png'});
fecom(cf,'ImWrite');comgui('iminfo',cf)

% Example of two channels in iiplot, with finish on same view
ci=iicom('curveload','gartid');iicom('ch20')
cingui('PlotWd',ci,'@OsDic(SDT Root)','FniiLeg');
RO=struct('ch',1:2, ...
    'FileName',{{'@PlotWd','Test','@cf.ua.ch','.png'}}, ...
    'RestoreFig',1, ... % reset at the end
    'ImWrite','ImWrite'); % Avoid the -ftitle
iicom(ci,'ImWrite',RO)
```

`comgui('ImFeplot')` returns a list of standard calls to options for image generation.

Pro

`iicom('ProFig')` shows or hides the properties figure.
`iicom(ci,'ProRefreshIfVisible')` refreshes the property figure when it is visible.
`iicom(ci,'ProInit')` reinits the property figure.

Show plot type

Specify the current axis type. The `iiplot` plot functions support a number of plot types which can be selected using the `Show` menu/command. From command line, you can specify the target axis with a `-cax i` option.

The main plot types are

- 2D ($f(x)$) plots are associated with the following buttons `Abs` (absolute value), `Pha` phase, `Phu` unwrapped phase, `Rea` real part, `Ima` imaginary part, `R&I` real and imaginary, `Nyq` Nyquist.
- 3D ($f(x,y)$) plots are `image`, `mesh`, `contour` and `surface`. `Show3D` gives time-frequency representation of the log of the abs of the signal displayed as and image. The `ua.yFcn` callback operates on the variable called `r3` and can be used for transformations (absolute value, phase, ...). Note that you may then want to define a colorbar see `iiplot PlotInfo`.

```
R1=d_signal('Resp2d'); % load 2d map
R1.PlotInfo= ii_plp('plotinfoTimeFreq -yfcn="r3=r3" type "contour"');
ci=iicom('curveinit','2DMap',R1);

% or
R1.PlotInfo={}; ci=iicom('curveinit','2DMap',R1);
ci=iicom('curveinit','2DMap',R1);
iicom('show3D -yfcn="r3=log10(abs(r3))" type "contour"')
```

- idcom specialized plots see **iiplot** TypeIDcom : mmi MMIF of Test, fmi forces of MMIF of Test, ami alternate mode indicator of Test, SUM of Test, CMIF of Test, sumi sum imaginary part of Test, pol poles in IdMain, fre freq. vs. damping in IdMain, rre real residue in IdMain , cre complex residue of IdMain, lny local Nyquist of Test (superposition around current pole), err Nyquist Error for current pole, Quality for all poles
- feplot plots.

SubSave, SubSet

SubSave*i* saves the current configuration of the interface in a stack entry TabInfo. This configuration can then be recalled with SubSet*i*. The TabInfo entry is also augmented when new curves are shown so that you can come back to earlier displays. SubSetI*i* selects an index in the TabInfo stack.

SubToFig

SubToFig copies the iiplot figure visualization to a standard matlab figure, thus allowing an easier handling to produce customized snapshots (see also **iicom** ImWrite). Reformatting is then typically performed with **comgui** objSet.

Command option -cf*i* forces the visualization output to figure *i*.

Command option leg*i* allows iiplot legend handling in the visualization. leg0 removes the legend, leg1 keeps it as in iiplot, leg2 transforms the iiplot legend in a standard matlab legend. The legend is removed by default.

Sub *plot init*

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.

Accepted command options are

- MagPha gives a standard subdivision showing a large amplitude plot and a small wrapped phase plot below.
- Iso gives a standard 2 by 2 subdivision showing four standard 2-D projections of a 3-D structure (this is really used by feplot).
- *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 with command option nd.

  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.
- step increments the deformation shown in each subplot. This is generally used to show various modeshapes in the same figure.
- Reset forces a reinit of all properties. For example SubMapha Reset.

TitOpt [,c]*i*, title and label options

Automated title/label generation options. TitOpt*i* sets title options for all axes to the value *i*. *i* is a 5 digit number with

- units corresponding to title. For modes [None,ModeNumber,Name].
- decades to xlabel 0 none, 1 label and units, 2 label.
- hundreds to ylabel 0 none, 1 label and units, 2 label.
- thousands to zlabel 0 none, 1 label and units, 2 label.
- 1e4 to legend/title switching.

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.

When checking the axes data (using `iicom Sub` command), one rebuilds the list of labels for each dataset using `iicom('chlab')`. This cell array of labels, stored in `ci.ua.chlab`, gives title strings for each channel (in rows) of datasets (in columns) with names given in `ci.ua.sList`. The label should start with a space and end with a comma. The dataset name is added if multiple datasets are shown. Not to display the curve name in the legend you can define and set `ci.ua.LegName = 0`, going back to default behavior is obtained by `ci.ua.LegName = 1`.

Modifying the `ci.IDopt.unit` value changes the unit assumed for identification but not the dataset units.

Titles and labels are not regenerated when using `ch` commands. If something is not up to date, use `iicom Sub` which rechecks everything.

Scale : xlin, xlog ...

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` option to change all axes: `iicom('xlog all')`.

`ytight[on,off,]` can be used to obtain tight scales on the `y` axis. The x axis is typically always tight. Automated `ztight` is not yet supported.

Limits : wmin, xlim, xmin, xmax, wmo, w0, ...

Min/max abscissa selection is handled using the fixed zoom (graphically use button). Accepted commands are

- `xlim` *min max* (or the legacy `wmin` *f1 f2*). For 2D plots, use `xlim` *xmin xmax ymin ymax* to allow selection of a 2D area.
- `xmin` *min* (or the legacy `wmin` *f1*)
- `xmax` *max* (or the legacy `wmax` *f1*)
- `wmo` allows a mouse selection of the minimum and maximum value (same as button).
- `w0` resets values (same as double click after hitting the button)

The limit value(s) can also be forwarded as last argument : `iicom('xlim',[min max])`.

When performing identification with `idcom` the fixed zoom corresponds to the working frequency range and corresponds to indices in `ci.IDopt(4:5)` (see `IDopt`, turn off with `idcom('Off')`). The index of the frequency closest to the specified min/max is used. When viewing general responses, the information for the abscissa limits is stored in the axis and is thus lost if that axis is cleared.

See also

`iiplot`, section 2.1 , `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.

clip [Start,Undo]

This command is used to eliminate faces not contained within the area that the user selects with a dragging box. ClipUndo clears the current axis and calls feplot to reinitialize the plot.

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.

Context menus are described in section 2.1.1 and section 4.4.1 .

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	?	list keyboard shortcuts

The list of the associated call is accessible using the call `sdt_table_generation('KeyPressTable.feplot')` or `sdt_table_generation('KeyPressTable.iiplot')`.

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 vector 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

Purpose

Refresh all the drawing `axes` of the `iiplot` interface.

Syntax

`iiplot`

Description

`iiplot` is used to scan through multiple sets of 1D (function of time or frequency) and 2D responses (functions of two variables) as discussed in `Type`. Section 2.1 gives an introduction to the use of `iiplot` and the companion function `iicom`.

- The data is stored in a `Stack` using one of the accepted `curve` formats.
  `iicom CurveInit` is the base command to add curves in the stack. You can also create a new `iiplot` axis using a curve data structure `Curve` (generated by `fe_curve` for example), simply calling `iiplot(Curve)`.
- Each `iiplot` axis (see `iicom Sub`,) can display some or all data sets in their stack. The selection of what is displayed is obtained using the `iicom IIx` commands or the `Variables` menu.
- `iiplot` with no arguments refreshes all the drawing axes.
- Plot `Type` 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` and the associated keys and toolbar buttons can be used to change selected channels.
- *Pole lines* for the indication of pole frequencies, or other lines to be shown (harmonics, thresholds, ...), are available for many plots. In general the information for these lines is stored as a `Curve.ID` field. The `IIplot:PoleLine` menu can be used to change how these lines appear. For identification (see `idcom`) `ci.Stack{'IdMain'}` pole lines are shown in white/black. `ci.Stack{'IdAlt'}` pole lines in red.

`ci : handle`

`ci=iiplot` returns a *SDT* `handle` to the current `iiplot` figure (2nd optional output argument is `XF`, a pointer to the curve stack, see section 2.1.2). You can create more than one `iiplot` figure with `ci=iiplot(FigHandle)`.

PlotInfo

Curves to be displayed can contain a `C1.PlotInfo` cell array of `stringTag,data`. An alternate form using matrix where the first column gives tag and the second the data is accepted if that matrix has at least two rows.

- `LineProp` specifies properties to be used as properties for lines. For example `C1=sdsetprop(C1,'PlotInfo','LineProp',{'LineWidth',2})`. This is checked at each display.
- `sub`, `show`, `scale` commands to be executed when initializing a display tab with `iicom Sub`.
- `ua.PostFcn` commands executed at the end of a refresh. This gives the user a chance to introduce modifications to the result of `iiplot`.
- `ua.TickFcn` commands executed whenever a mouse zoom is done, see `TickFcn`.
- `ua.axProp` is a cell array containing properties to be applied with an `comgui objSet` command.
- `ColorBar` is a cell array containing properties to be used to generate a colorbar. See `fecom ColorBar` for more details. For example

```
C1=d_signal('Resp2D');
C1=sdsetprop(C1,'PlotInfo','ColorBar',{'YAxisLocation','left'});
iicom('curveinit','2D',C1);
```

- `LDimPos` specifies the dimension used to generate the label on the response axis (y for $f(x)$, z for $f(x,y)$).

The `ii_plp('PlotInfo',C1)` command provides default values for classical configurations.

Type

- 2D ($f(x)$) plots are associated with the following buttons and `iicom Show` commands `Abs` (absolute value), `Pha` phase, `Phu` unwrapped phase, `Rea` real part, `Ima` imaginary part, `R&I` real and imaginary, `Nyq` Nyquist.
- 3D ($f(x,y)$) plots are `image`, `mesh`, `contour` and `surface`. For this plots `ua.XDimPos` should give the positions of dimensions associated with the x and y variations. Proper `.PlotInfo` can be generated with `ii_plp('PlotInfo2D -type "contour"',C1)`.

DimPos and channel

When displaying multi-dimensional curves as 2D plots $f(x)$, the abscissa x is taken to be the first dimension declared in the `C1.DimPos` field (with a default at 1).

When displaying as 3D ($f(x,y)$) plots, the x, y are taken to be the first two dimensions declared in the C1.DimPos field (with a default at 1,2). You can then flip the positions in the plot axis by setting `ci.ua.XDimPos=[2 1]`.

Channels are indices for remaining dimensions.

The y (z for 3D) axis label is built using the `C1.DimPos(2)` dimension unless the curve contains a `LDimPos` entry.

TypeIDcom

Specialized plots for `idcom` are

- *Local Nyquist plots* (initialized by `show lny`) show a comparison of `Test` (measured FRFs) and `IdFrf` (identified model) in a reduced frequency band
$$\left[\begin{array}{cc} \omega_j(1-\zeta_j) & \omega_j(1+\zeta_j) \end{array} \right] \tag{10.29}$$
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.

- *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 `'Test'` set.

- *Pole locations in the complex plane* (initialized by `show pol`).

- *Poles shown as damping vs. frequency* are initialized by `show fre`.

- *Position of residues in the complex plane* are initialized by `show cre`. This plot 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` *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 `ci.Stack{'Test'}.xf - ci.Stack{'IdMain'}.xf` divided by RMS response of `ci.Stack{'Test'}`) is shown as a dashed line with `+` markers and a normalized response level (RMS response of `ci.Stack{'Test'}`) as a dashed line with `x` markers.

  Normalized errors should be below 0.1 unless the response is small. You can display the error using the nominal sensor sort with `ci.Stack{'IdError'}.sort=0` and with increasing error using `sort=1`.

- `Quality` *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

$$J_{ij}(\Omega) = \sum_{ij \text{ measured}, k \in \Omega} |\hat{H}_{ij}(s_k) - H_{ij}(s_k)|^2 \tag{10.30}$$

and the log-least-square cost function by

$$J_{ij}(\Omega) = \sum_{ij \text{ measured}, k \in \Omega} |log \left| \frac{\hat{H}_{ij}(s_k)}{H_{ij}(s_k)} \right| |^2 \tag{10.31}$$

For sets `xf` and `xe` stored using the `xf` format (see page 214), `ii_cost` computes both those costs which are used in identification and model update algorithms (see section 3.2.3).

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.

The high level implentation of tools provided by ii_macis interfaced in the dock CoShape. A summary of typical applications is given in section 3.2 and examples in the d_cor demo.

You can also use low level calls to just display a figure or a table, as illustrated by the expamples below.

Vector correlations are *SDT* objects which contain deformations, see va, typically given at test sensors. For criteria using model mass or stiffness matrices see m. Other details about possible fields of VC 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.

```
[model,sens,ID,FEM]=demosdt('demopairmac'); % Sample data

cf=comgui('guifeplot-reset',2); % force feplot in figure(2);
cf.model=model;  % Display FEM (contains topology correl in Test)

VC=ii_mac(ID,FEM,'sens',sens,'mac plot -cf1');
```

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

```
[cf,def_fem,res_test]=demosdt('demo gartte cor plot');
Sens=fe_case(cf.mdl,'sens');
figure(1); subplot(221); VC=ii_mac(1);% Make figure(1) current so that ii_mac uses it
ii_mac(VC,res_test,def_fem,'labela','Test','labelb','FEM', ...
          'sens',Sens,'Mac Pair Plot');
subplot(212);ii_mac(VC,'comac'); % set new axis and display other criterion
subplot(222); ii_mac(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)`.

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 link deformations shown in a `feplot` figure to a MAC plot using

```
[model,sens,ID,FEM]=demosdt('demopairmac');
cf=feplot(model);
cf.def(1)=ID;  % display test as first def set
cf.def(2)=FEM; % display FEM as second def set
% overlay & show interactive MAC in fig 1:
figure(1);clf;fecom('show2def-mac1')
```

Main commands

`Options ...  [Plot,Table,Tex,Thtml]`

By default, the commands plot the result in a figure. Options valid for all commands are

- `plot` generates figure in the current axis. You can use `figure` and `subplot` to set the current axis.
- `Table` generates a text output
- `Tex` generates a format suitable for direct inclusion in LaTeX
- `Thtml` creates and open an HTML file in the MATLAB browser.

Data fields

Data fields are defined using name, value pairs.

- 'cpa',dataAsCols sets shapes . But calls with data structures are preferable, see va.
- 'sens',sens sets sensor observation matrix, see sens.
- 'labela','name' sets the name of data set A. Typical values are Test, FEM, ...
- 'inda',ind selects vectors given by ind when computing a criterion. For example, rigid body modes are often not included in correlation. Thus 'indb',7:20 would skip the first 6 modes.
- 'SubDofInd',ind allows the selection a subset of sensors when computing correlation criteria.

MAC [,M] [,PairA,PairB,AutoA, ...] ...

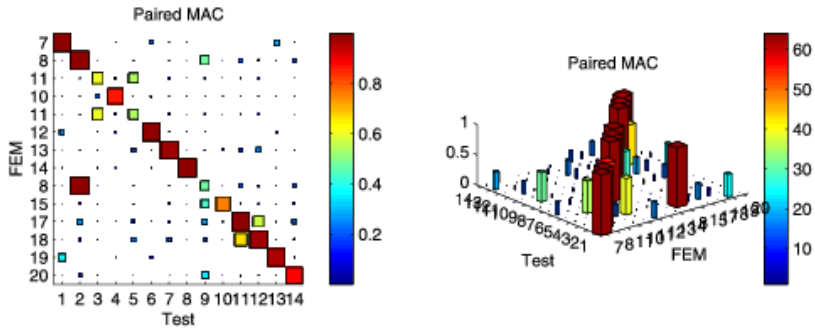
The Modal Assurance Criterion (MAC) [4] is the most widely used criterion for vector correlation (mainly because of its simplicity).

The MAC is the correlation coefficient of vector pairs in two vector sets cpa and cpb defined at the same DOFs (see ii_mac va for more details). 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

$$\mathrm{MAC}_{jk} = \frac{|\{c\phi_{\mathtt{id}j}\}^H \{c\phi_k\}|^2}{|\{c\phi_{\mathtt{id}j}\}^H \{c\phi_{\mathtt{id}j}\}||\{c\phi_k\}^H \{c\phi_k\}|} \tag{10.32}$$

For two vectors that are proportional the MAC equals 1 (perfect correlation). Values above 0.9 are generally considered as well correlated. Values below 0.6 should be considered with much caution (they may or may not indicate correlation).

The commands and 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.



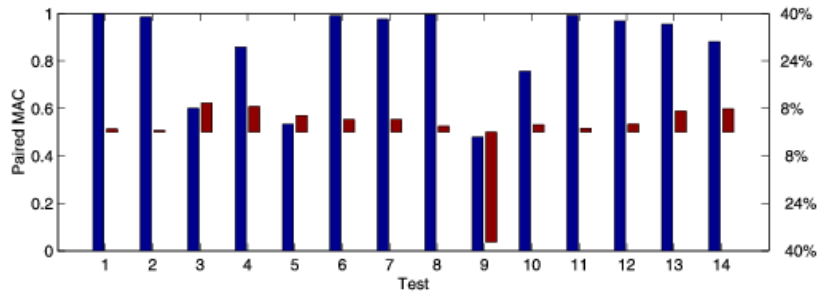

```
[model,sens,ID,FEM]=demosdt('demopairmac');

if ishandle(1);close(1);end;figure(1);
VC=ii_mac(1,ID,FEM,'sens',sens,'mac paira table')
ii_mac(VC,'mac paira plot');
```

The basic MAC shows vector pairs for all vectors in the two sets. Specific command options are

- `MacM` should be used when a mass is provided, see `MacM`
- `MacPairA` command seeks for each vector in `cpa` (`cpb` with `PairB`) the vector in `cpb` (`cpa`) that is best correlated. `MacPairB` pairs against `cpb` vectors.
- `MacAutoA` 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).
- `-combine`*`val`* allows orthogonal linear combinations of vectors whose frequencies are closer than *`val`* relatively. This is meant for cases with very closely spaced modes where subspaces rather than individual vectors should be compared.
- `Error` 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`). Typical calls can be found in `gartco` example.

  By default this command displays a plot similar to the one shown below where the diagonal of the paired MAC and the corresponding relative error on frequencies are shown. For text output see general command options.



This is an example of how to use of the MACError command. In this example, the only significant errors are associated with mode crossing so that the .Combine gives a nearly perfect coerrelation.

```
[model,sens,ID,FEM]=demosdt('demopairmac');
if ishandle(1);close(1);end;figure(1);
VC=ii_mac(1,ID,FEM,'sens',sens)
ii_mac(1,'SetMAC',struct('Pair','A','MacPlot','do'))
ii_mac(1,'macerror table',struct('MinMAC',.6,'Df',.2,'Combine',.1));
ii_mac(1,'SetMAC',struct('MacError','do'))
```

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 (10.32)). 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. You do not have to determine modal masses (see id_rm) to compute a MAC.
- 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 natures 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 vectors defined at all DOFs, the MAC is a poor criterion and you should really use its mass weighted counter part. 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.

- In 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 non weighted MAC. In such cases, you should really prefer the mass weighted MAC for correlation.

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 used with command option text prints an output of the form

```
   Test         1   2   3   4   5   6   7   8
    FEM         7   8  11  10  11  12  13  14
 Sensor Mean
    All    87 100  99  60  86  53 100  98 100
  1112z    88 100  99  59  90  62 100  98 100
  1301z    89 100  99  62  90  64 100  98 100
  1303z    90 100  98  66  90  66 100  98 100
```

where the indices for the vectors used in the pairing are shown first, then followed by the initial mean MAC and MAC associated to each pair. The following lines show the evolution of these quantities when sensors are removed. Here sensor 1112z has a strong negative impact on the MAC of test mode 5.

The sensor labels are replaced by sensor numbers if the sensor configuration sens is not declared.

By default the MACCO command outputs a structure in which field .data contains in its first column the sensor or index removed and the resulting MAC evolution of paired modes in the following columns. The field .xlabel contains the sensor labels or indices.

Command option plot will plot in the ii_mac figure the MAC evolutions as function of the sensors removed. Command option text will output the result as text.

This is an example of how to use of the MACCO command

```
% To see the result
 [model,sens,ID,FEM]=demosdt('demopairmac');
 figure(1);clf;VC=ii_mac(1);
 ii_mac(VC,ID,FEM,'sens',sens, ...
  'inda',1:8, ... % Select test modes to pair
  'macplot')
% See sensors for each mode
```

```
r1=ii_mac(VC,'inda',1:8,'MacCo',struct('Table',1,'ByMode',1,'N',5));
% See sensors improving mean modes
r2=ii_mac(VC,'inda',1:8,'MacCo',struct('Table',1,'N',5));

% Numeric values stored in r1 and r2.
```

MacM ...

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

$$\text{MAC-M}_{jk} = \frac{|\{\phi_{jA}\}^T [M] \{\phi_{kB}\}|^2}{|\{\phi_{jA}\}^T [M] \{\phi_{jA}\}||\{\phi_{kB}\}^T [M] \{\phi_{kB}\}|} \tag{10.33}$$

called with ii_mac(... 'm',m,'MacM Plot'). 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. When importing an external reduced mass matrix, just define the mass as shown above, when using *SDT*, see the ii_mac 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.

COMAC [,M][,A,B][,N][,S][,E] [,sort]

The COMAC command supports three correlation criteria (N nominal, S scaled and E 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 options (with no option the command computes all three). Sensors are given in the nominal order or sorted by decreasing COMAC value (sort command option).

These criteria assume the availability of paired sets of sensors. The COMAC commands thus start by pairing modes (it calls MacPair or MacMPair) to pair vectors in cpb to vectors in cpa. The B command option 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

$$\text{COMAC}_l = 1 - \frac{\left\{\sum_j^{NM} |c_l\phi_{jA} c_l\phi_{jB}|\right\}^2}{\sum_j^{NM} |c_l\phi_{jA}|^2 \sum_j^{NM} |c_l\phi_{jB}|^2} \tag{10.34}$$

which is 1 minus the definition found in [56] 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)

$$\left\{\widehat{c\phi_{jB}}\right\} = \{c\phi_{jB}\}\,\text{MSF}_j = \{c\phi_{jB}\}\,\frac{\{c\phi_{jB}\}^T\{c\phi_{jA}\}}{\{c\phi_{jB}\}^T\{c\phi_{jB}\}} \tag{10.35}$$

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 [57], is given by

$$\text{eCOMAC}_l = \frac{\sum_j^{NM} \left\| \left\{\widetilde{c_l\phi_{jA}}\right\} - \left\{\widehat{c\phi_{jB}}\right\} \right\|}{2NM} \tag{10.36}$$

where the comparison is done using modeshapes that are vector normalized to 1

$$\left\{\widetilde{c_l\phi_{jA}}\right\} = \{c\phi_{jA}\} \,/\, \|c\phi_{jA}\| \tag{10.37}$$

This is an example of how to use of the COMAC command

```
[model,sens,ID,FEM]=demosdt('demopairmac');

figure(1);clf;VC=ii_mac(1);
ii_mac(VC,ID,FEM,'sens',sens,'comac plot')
ii_mac(VC,'comac table');
```

```
POC [,Pair[A,B]] ...
```

The orthogonality conditions (6.96) 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

$$\mu_{jk} = \{\phi_{jA}\}^T [M] \{\phi_{kB}\} \qquad \kappa_{jk} = \{\phi_{jA}\}^T [K] \{\phi_{kB}\} \tag{10.38}$$

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` options are available for `POC` just as for the MAC.

`Rel [,scaled][,m]`

For scaled matched modeshapes, the **relative error**

$$e_j = \frac{\| \{c\phi_{jA}\} - \{c\phi_{jB}\} \|}{\| \{c\phi_{jA}\} \| + \| \{c\phi_{jB}\} \|} \tag{10.39}$$

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 (10.39).

For uncalled 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

$$\text{MSF}_j = \frac{\{c\phi_{jA}\}^T \{c\phi_{jB}\}}{\{c\phi_{jB}\}^T \{c\phi_{jB}\}} \tag{10.40}$$

The `RelScale` command calls `MacPair` to obtain shape pairs, multiplies shapes in set B by the modal scale factor and plots the result of (10.39).

With the `M` option, the `MacPairM` is used to obtain shape pairs, kinetic energy norms are used in equations (10.39)-(10.40).

This is an example of how to use the Rel command

```
[model,sens,ID,FEM]=demosdt('demopairmac');
ii_mac(ID,FEM,'sens',sens,'rel');
```

VC

The following sections describe standard fields of VC 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,vb,sens

ii_mac uses two data sets referenced in VC.va and VC.vb and extracts shapes at sensors using the get_da_db command shown below. All standard input formats for shape definition are accepted

- FEM result with .def and .DOF fields, see section 7.8 .
- Shapes at DOFs or pole residue with .res and .po fields (see section 5.6)
- Response data with .w and .xf fields
- simple matrix with rows giving DOFs and columns shapes. These will be stored in the va.def field, called cpa which stands for $[c]\{\phi_a\}$ since these vectors typically represent the observation of modeshapes at test sensors, see section 5.1 . A typical call would thus take the form

```
FigHandle=figure(1);
ii_mac(FigHandle,'cpa',shapes_as_col,'labela','Test', ...
    'cpb',shape2, ... % Define vb
    'mac');  % define command
```

sens, when defined (see section 4.6 for the generation of sensor configurations), does not use the results defined in VC.va but their observation given by VC.sens.cta*VC.va.def (same for VC.vb).

The illustration below uses a typical identification result ID, a FEM result FEM and observes on sensors.

```
[model,sens,ID,FEM]=demosdt('demopairmac -open')
figure(1);[r1,VC]=ii_mac(ID,FEM,'sens',sens, ...
 'indb',7:20,'mac plot');
[da,db]=ii_mac(VC,'get_da_db')
```

The `da.def` and `db.def` fields are always assumed to be observed at the same sensors (correspond to the `cpa`, `cpb` fields if these are defined).

To support expansion, `cpa` is defined at DOFs and `cpb` at sensors, `ii_mac` calls `fe_exp` to expand `cpb` on all DOFs.

`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 6.2.4).

`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 $T^T M T$) on the subspace

$$[T] = \left[\tilde{T}\right]\left[c\tilde{T}\right]^{-1} \quad \text{with} \quad [K]\left[\tilde{T}\right] = [c]^T \tag{10.41}$$

In cases where the model is fixed $[K]$ is non-singular and this definition is strictly equivalent to static/Guyan condensation of the mass [16]. When the structure is free, $[K]$ should be replaced by a mass shifted $[K]$ as discussed under the `kd` field above.

T

Reduced basis expansion methods were introduced in [16]. Static expansion can be obtained by using T defined by (10.41).

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 3.2

ii_mmif

Purpose

Mode indicator functions and signal processing.

Syntax

```
OUT =  ii_mmif('command',IN,'waitbar')
ci=iiplot; ii_mmif('command',ci,'CurveName')
```

Description

This function supports all standard transformations of response datasets in particular mode indicator functions and signal processing.

With data stored in a `iiplot` figure, **from the GUI**, open the Stack tab of the property figure (accessible through `iicom('CurtabStack')` or by clicking on) then select `Compute ...` in the context menu to transform a given dataset. This has the advantage of allowing interactive changes to signal processing results, see section 2.1.7 .

From the command line, use `ii_mmif('command',ci,Curve)` (where `ci` is a handle referring to `iiplot` figure). `Curve` can be a string defining a curve name or a regular expression (beginning by `#`) defining a set of curves. One can also give some curve names as strings in a cell array. Without output argument, computed `mmif` is stored in the stack with name `mmif(`*`CurveName`*`)`. Accepted command options are

- `-reset` to compute a `mmif` which has already been computed before (otherwise old result is reused). The existence is based on the name in the `iiplot` stack.
- `-display` displays the result in the associated `iiplot` figure.

```
ci=iicom('curveload','gartid'); % load curve gartid example
ii_mmif('mmif',ci,'Test');      % compute mmif of set named Test
iicom('iixonly',{'mmif(Test)'});% display result
```

When used with `idcom`, the `Show ...` context menu supports the automated computation of a number of transformations of `ci.Stack{'Test'}`. These mode indicator functions 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.

With data structures not in `iiplot` use `mmif=ii_mmif(`*`command`*`,Curve)`. Use command option `-struct` to obtain output as curve data structure.

```
ci=iicom('curveload','gartid'); % load curve gartid example
R1=ci.Stack{'Test'};            % get Test dataset in variable R1
R2=ii_mmif('mmif-struct',R1);   % compute mmif
```

MMIF

The Multivariate Mode Indicator Function (MMIF) (can also be called using `iicom Show mmi`) was introduced in [58]. Its 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

$$q(s,\{u\}) = \frac{\{u\}^T \left[\mathrm{Re}(H)^T \mathrm{Re}(H)\right]\{u\}}{\{u\}^T \mathrm{Re}([H^H H])\{u\}} = \frac{\{u\}^T [B]\{u\}}{\{u\}^T [A]\{u\}} \tag{10.42}$$

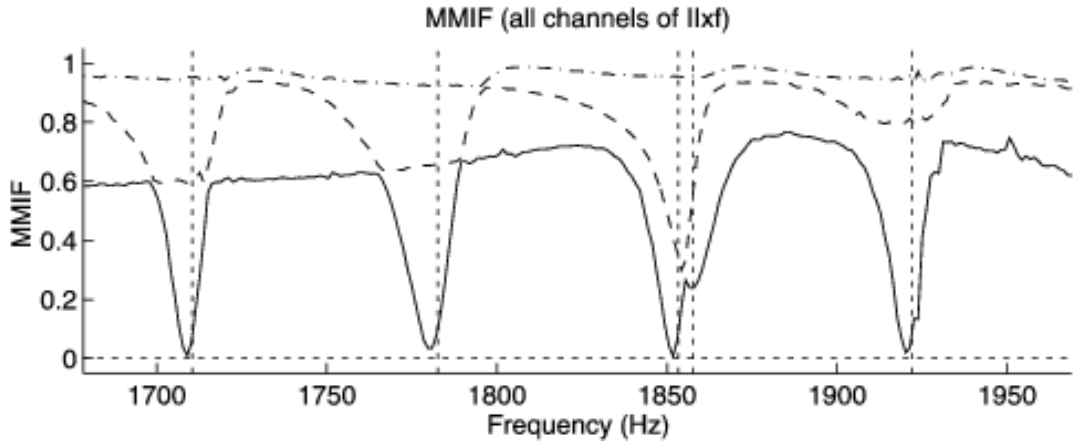
For structures that are mostly elastic (with low damping), resonances are sharp and have properties similar to those of isolated modes. The MMIF (q) thus drops to zero.

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 extreme 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 for 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 a minimum 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 [9].



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 [59]. These forces are the second optional output argument `ua`.

CMIF

The Complex Mode Indicator Function (CMIF) (can also be called using `iicom Show cmmi`, see [60] 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

$$[H(s)]_{NS\times NA} = [U]_{NS\times NS}\,[\Sigma]_{NS\times NA}\left[V^H\right]_{NA\times NA} \tag{10.43}$$

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

$$q(s) = 1 - \frac{|\mathrm{Im}(H(s))||H(s)|^T}{|H(s)||H(s)|^T} \tag{10.44}$$

which has been historically used in force appropriation studies [59]. 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 amif`).

SUM, SUMI, SUMA

Those functions are based upon the sum of data from amplitude of sensors for a given input. One can specify dimensions affected by the sum using command option `-dim i` (*i* is one ore more integers).

`SUM`,

$$S(s,k) = \sum_j \|H_{j,k}(s)\|^2 \tag{10.45}$$

is the sum of the square of all sensor amplitude for each input.

`SUMI`,

$$S(s,k) = \sum_j \mathrm{Im}(H_{j,k}(s))^2 \tag{10.46}$$

is the sum of the square of the imaginary part of all sensors for each input.

`SUMA`,

$$S(s,k) = \sum_j \|H_{j,k}(s)\| \tag{10.47}$$

is the sum of the amplitude of all sensors for each input.

Those functions are sometimes used as mode indicator functions and are thus supported by `ii_mmif` (you can also call them using `iicom Show sumi` for example).

NODEMIF

Undocumented.

Signal processing

Following commands are related to signal processing. Section section 2.1.7 illustrates the use of those functions through `iiplot`.

Integrate, DoubleInt, Vel, Acc

- `Integrate` integrates the frequency dependent signal

$$I_{j,k}(s) = \frac{H_{j,k}(0)}{s^2} + \frac{H_{j,k}(s)}{s}. \tag{10.48}$$

- `DoubleInt` integrates twice the frequency dependent signal

$$I2_{j,k}(s) = \frac{H_{j,k}(0)}{s^3} + \frac{H_{j,k}(s)}{s^2}. \tag{10.49}$$

- `Vel` computes the velocity (first derivative) of the signal. For a frequency dependent signal

$$V_{j,k}(s) = s \cdot H_{j,k}(s). \tag{10.50}$$

  For a time dependent signal, finite differences are used

$$V_{j,k}(t_n) = \frac{H_{j,k}(t_{n+1}) - H_{j,k}(t_n)}{t_{n+1} - t_n}. \tag{10.51}$$

  $V_{j,k}(t_{end})$ is linearly interpolated in order to obtain a signal of the same length.

- `Acc` computes the acceleration (second derivative) of the signal. For a frequency dependent signal

$$A_{j,k}(s) = s^2 \cdot H_{j,k}(s). \tag{10.52}$$

  For a time dependent signal, finite differences are used

$$A_{j,k}(t_n) = \frac{h_n \cdot (H_{j,k}(t_{n+1}) - H_{j,k}(t_n)) - h_{n+1} \cdot (H_{j,k}(t_n) - H_{j,k}(t_{n-1}))}{h_{n+\frac{1}{2}} \cdot h_n \cdot h_{n+1}}, \tag{10.53}$$

with $h_{n+1} = t_{n+1} - t_n$ and $h_{n+\frac{1}{2}} = \frac{h_n + h_{n+1}}{2}$.

$A_{j,k}(t_{end})$ and $A_{j,k}(t_1)$ are linearly interpolated in order to obtain a signal of the same length.

FFT, FFTShock, IFFT, IFFTShock

Computes the Discrete Fourier Transform of a time signal. FFT normalizes according to the sampling period whereas FFTShock normalizes according to the length of the signal (so that it is useful for shock signal analysis).

IFFT and IFFTShock are respectively the inverse transform.

Accepted command options are

- -nostat to remove static component (f=0) from fft response.
- -newmark to shift frequencies of computed time integration with a mean acceleration Newmark scheme ($\gamma = 0.5, \beta = 0.25$) in order to correct the periodicity error $\frac{\Delta T}{T} = \frac{\omega^2 h^2}{12}$. This correction is especially true for low frequencies. Command option -newmark-beta*val* allows specifying another value of β, using the general shift value $\frac{\Delta T}{T} = \frac{1}{2}(\beta - \frac{1}{12})\omega^2 h^2$.
- tmin *value*, tmax *value*, fmin *value*, fmax *value* to use parts of the time trace or spectrum.
- zp *value* is used to apply a factor *value* on the length of the signal and zero-pad it.
- -window *name* is used to apply a window on the time signal. Use fe_curve('window') to get a list of implemented windows. For windows with parameters, use double quotes. For example R1_FFT=ii_mmif('FFTShock -struct -window "Exponential 10 20 100"',R1).
- -display force display in iiplot after computing

```
[model,def]=fe_time('demobar10-run');
R1=ii_mmif('FFT-struct -window "hanning" wmax 400',def);
% To allow interaction
ci=iiplot;ci.Stack{'curve','def'}=def;
ii_mmif('FFT-struct -window "hanning" fmax 400 -display',ci,'def');
iicom('CurtabStack') % Show the property figure
```

BandPass

ii_mmif('BandPass fmin *fmin* fmax *fmax*') Performs a true band pass filtering (i.e. using **fft**, truncating frequencies and go back to time domain with **ifft**) between *fmin* and *fmax* frequencies.

OctGen, Octave

filt=ii_mmif('OctGen *nth*',f) computes filters to perform a $\frac{1}{nth}$-octave analysis.

As many filters as frequencies at the $\frac{1}{nth}$-octave of 1000 Hz in the range of **f** (vector of frequencies) are computed. Each band pass filter is associated to a frequency f_0 and a bandwidth Bw depending on f_0. Filters are computed so that their sum is almost equal to 1. Filter computed are, for each f_0 :

$$H(f, f_0) = \frac{1}{1 + (\frac{1}{Bw(f_0)} . \frac{f^2 - f_0^2}{f})^6} \tag{10.54}$$

With command option plot, filters are plotted.

ii_mmif('Octave *nth*',ci) performs the $\frac{1}{nth}$ octave analysis of active curve displayed in iiplot figure.

The $\frac{1}{nth}$ octave analysis consists in applying each filter on the dataset. Energy in each filtered signal is computed with $10log(S)$ (where S is the trapezium sum of the filtered signal, or of the square of the filtered signal if it contains complex or negative values) and associated to the center frequency of corresponding filter.

See also

iiplot, iicom, idopt, fe_sens

ii_plp

Purpose

Pole line plots and other plot enhancement utilities.

Syntax

```
ii_plp(po)
ii_plp(po,color,Opt)
```

Description

plp

Generation of zoomable vertical lines with clickable information.

`ii_plp(po)` will plot vertical dotted 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`).

.ID PoleLine Call from iiplot

When displaying a curve in `iiplot`, one can generate automatic calls to `ii_plp`. `Curve.ID` field can be used to generate automatically vertical lines in iiplot. It is a cell array with as many cell as line sets. Each cell is a data structure defining the line set. Following fields can be defined:

- `.po` can be a column vector defining abscissa of vertical lines. It can also be a string, possibly depending on the displayed curve `XF1` and the channel through variable `ch` to be evaluated to define the `ro.po` vector, for example `'r1.po=XF1.Y2(:,ch);'`.
- `.LineProp` is optional. One can specify some MATLAB line properties in this field as a cell array `{'prop1', value1, 'prop2', value2, ...}`, for example `{'LineStyle',':','color','r'}`.

  When using line sequencing, it is preferable to set the property using the line object tag `now`. Thus

```
R1=sdsetprop(R1,'PlotInfo.ua.axProp', ...
    '@now',{'LineStyle','--','color','k','marker','none'});
```

- `.marker` supports alternative strategies for line generation. `horizontal` assumes `.po` to given horizontal values. `xy` allows `.po` to contain a curve. If defined `.po.MainDim='y'` the curves are assumed to be $x = f(y)$ rather than the traditional $y = f(x)$.
- `.name` is used to generate a text info displayed when the user clicks on the line.
- `.unit`(obsolete) is used for Hz vs. rad/s unit conversion. 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)`.
- `.format` an integer that 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.

Legend

Dynamic multi-line legend generation used by `iiplot` and `feplot`.

`ii_plp('legend',ga,prop)` with properties a cell array detailed with in `comgui def.Legend` (typical legend generation associated with FEM solutions).

- `'set','legend -corner`*x y*`'` gives the position of the legend corner with respect to the current axis. `-reset` option deletes any legend existing in the current axis.
- `'string',StringCell` cell array of strings with one per line of legend. Line specific text properties can be given in second column of StringCell.
- `'PropertyName',PropertyValue` additional properties to be set on the created text.

`ii_plp('legend -corner .01 .01 -reset ',ga,ua,StringCell,legProp)` is an older format found in some calls, with `ga` handle to the axis where the axes is to be placed, see `gca`. `ua` if not empty provides additional properties `.legProp, .Corner`.

PlotSubMark

Generate subsampled markers.

spy

`ii_plp('spy',k);` Generates a `spy` plot with color coding associated with the non-zero element values of matrix `k`.

- `unsymm` is used to force non symmetric plots.
- `threshold` is used force small terms to zero.
- `msize`*`val`* allows specifying the plot `MarkerSize` to *`val`*
- `-nopbar` avoids customizing the figure `PlotBoxAspectRatio` to respect the matrix one.

To perform block-wise spy plots of a single matrix, it is possible to provide matrix `k` as a structure with fields

- `K` the matrix to spy
- `ind` a cell array of disjoint sets of indices standing for a sequenced block-wise reodering of matrix K.
- `indC` (optional) to provide a different ordering for columns than for lines (following `ind`), activated for the `unsymm` case. It can be useful to display rectangular matrices.

TickFcn

SDT implements a general mechanism for enhancing the basic dynamic tick label generation of MATLAB. This allows placement of strings are proper locations on an axis. `ii_plp('TickFcn')` list predefined ticks.

This functionality is not fully documented and you are encouraged to look-up the source code. SDT generated plots expect the following fields in the axis userdata `ua.TickInfo` for data and `ua.TickFcn` for the callback. A sample usage would be

```
C1=struct('X',{{num2cell(2:4)' 2}},'Xlab',{{'x','y'}}, ...
 'Y',(1:3)')
figure(1);plot(1:3,C1.Y);ii_plp('tickXCell',C1,gca);
C1=ii_plp('tickXCell',C1); %Defines the PlotInfo
iiplot(C1);
```

ColorMap[,RO]

FEM oriented color maps.

Predefined maps can be directly called and will apply to the current figure. feplot assignment can be performed by nesting the ColorMap call in a call to fecom. fecom('colormapjet(5)') thus generates a map with 5 colors and grey level bands on the current feplot figure. This is called using

Sample colormaps are featured in the example below,

```
% Example of colormaps provided by ii_plp
figure(1);h=mesh(peaks(300));
set(h,'edgecolor','none','facecolor','interp');
ii_plp('ColormapBandjet(5)')
ii_plp('ColormapFireIce 20')
ii_plp('ColormapSamcef')
```

An exhaustive list can be obtained using ii_plp('ColorMap') with no argument. This will open the tag list for colormap thus showing the currently available maps.

- ii_plp('ColorMapWCenter Thres.1',jet(20)) uses the map given as second argument with a symmetric clim and a white band for values below the specified Thres.

In a more general context, one can define in any MATLAB figure a custom colormap with custom and unevenly spread thresholds by providing a structure in second argument with fields

- map the chosen map in rgb format.
- cval a vector of values at which colors switches. The color limits CLim properties of the figure current axis will be set to the extrema of cval. It is thus recommended to use clean min max values. Color distribution is performed sequentially, so that only one color per cval step is used. It is thus recommended to use a map with N-1 colors, N being the vector size.
- refine, optional. This is used to provide the colormap refinement needed to place color switches accurately. The default is set to 100.
- bSplit, optional. This is used to add black split lines between colors, with a specified thickness.
- Band, optional. This is used to add a darkening nuance to each color step.
- cf, optional. To provide a feplot or figure handle, by default the current figure is taken.

```
% Custom colormap setting using ii_plp
demosdt('demoubeam')
cf=feplot;
fecom(cf,'colordataa')
fecom(cf,'colorscale Unit 1e3');
fecom(cf,'colorbaron')
ii_plp('colormap',struct('map',[1 1 1;jet(7)], ...
 'cval',[0:4:10 11:2:21],'Band',0,'refine',10,'bSplit',2))
```

Cb

Callbacks for `comgui objSet` properties of colorbar. Accepted options are

- `No` north (main location), `eot` east outside top.
- `String` label of colorbar
- `map` color map command.
- `cf` figure number

```
figure(1);clf;mesh(peaks);
ii_plp('cbNo',struct('String','Z', ...
   'map','ii_plp(''ColormapBandjet(5)'')', ...
   'cf',1));
```

See also

ii_pof, idopt, iiplot, iicom

ii_poest

Purpose

Identification of a narrow-band single pole model.

Syntax

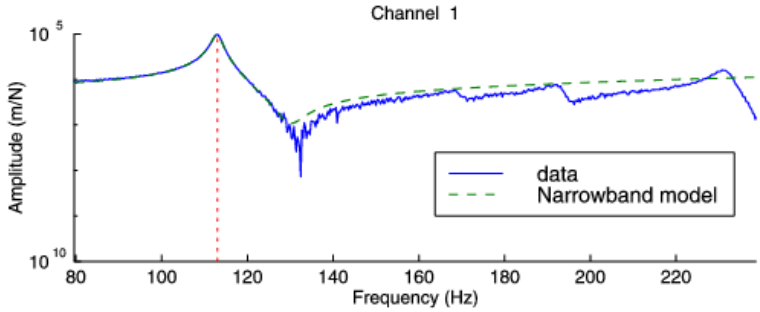
```
idcom('e')
[res,po]= ii_poest(ci.Stack{'Test'},opt)
```

Description

ii_poest (idcom e command and associated button in the idcom GUI figure, see section 2.8) 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, Test and IdFrf are automatically overlaid as shown in the plot above) and based on the message passed

```
>> ci=idcom;iicom(ci,'CurveLoad','gartid');
>> idcom('e .01 16.5');
>> disp(ci.Stack{'IdAlt'}.po)
  1.6427e+001  1.3108e-002
LinLS: 5.337e-001, LogLS 5.480e-001, nw 18
 mean(relE) 0.00, scatter 0.47 : acceptable
```

```
Found pole at 1.6427e+001   1.3108e-002
% manual call would be [res,po]=ii_poest(ci.Stack{'Test'},[.01 16.5]);
```

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 IdAlt to IdMain) 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

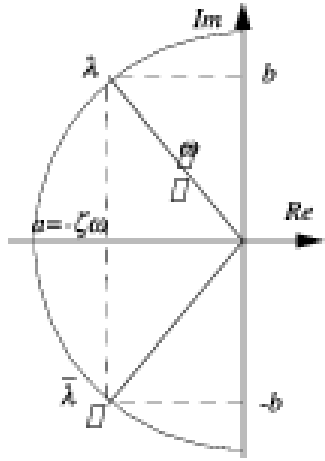
```
[pob] = ii_pof(poa,DesiredFormatNumber)
[pob] = ii_pof(poa,DesiredFormatNumber,SortFlag)
```

Description

The *Structural Dynamics Toolbox* deals with real models so that poles are either real or come in conjugate pairs

$$\{\lambda, \bar{\lambda}\} = \{a \pm ib\} = \{-\zeta\omega \pm \omega\sqrt{1-\zeta^2}\} \tag{10.55}$$

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

$$\texttt{po} = \left\{ \begin{array}{c} \lambda_1 \\ \bar{\lambda}_1 \\ \vdots \\ \lambda_{Re} \\ \vdots \end{array} \right\} \quad \text{for example} \quad \begin{array}{l} \texttt{po=[-0.0200 + 1.9999i} \\ \texttt{\quad -0.0200 - 1.9999i} \\ \texttt{\quad -1.0000]} \end{array} \tag{10.56}$$

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

$$\texttt{po} = \left[\begin{array}{cc} a & b \\ \vdots & \vdots \end{array} \right] \quad \text{for example} \quad \begin{array}{l} \texttt{po=[-0.0200 1.9999} \\ \texttt{\quad -1.0000 0.0000]} \end{array} \tag{10.57}$$

Format 3: frequency ω and damping ratio ζ

$$\texttt{po} = \left[\begin{array}{cc} \omega_1 & \zeta_1 \\ \vdots & \vdots \end{array} \right] \quad \text{for example} \quad \begin{array}{l} \texttt{po=[2.0000 0.0100} \\ \texttt{\quad -1.0000 1.0000]} \end{array} \tag{10.58}$$

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 5.6).

See also

idcom, id_rc, ii_plp

lsutil

Purpose

Level set utilities.

Syntax

```
model=lsutil('cut',model,li,RO)
def=lsutil('gen',model,li)
lsutil('ViewLs',model,li)
```

Description

`lsutil` provides a number of tools for level-set creation and manipulation.

Some commands return the model structure while others return the value of the level-set. Plot outputs are also available.

Available `lsutil` commands are

`edge`

`gen`

Level-set computation.

rect(lx,ly,xc,yc,alpha);

circ(rc,xc,yc);

box(lx,ly,lz,xc,yc,zc,nx,ny,nz);

sphere(rc,xc,yc,zc));

cyl(xc,yc,zc,nx,ny,nz,rc,z0,z1);

`cut`

Accepted options are

- `.doCut` start by dividing elements
- `.tolE` fractional distance to edge end considered used to enforce node motion.
- `.Fixed` nodes that should not be moved.

Here a first example with placement of circular piezo elements

```
RO=struct('dim',[400 300 8],'tolE',.3);
[mdl,li]=ofdemos('LS2d',RO);lsutil('ViewLs',mdl,li);
li{1} % Specification of a circular level set
RO.doSplit=1; mo3=lsutil('cut',mdl,li,RO);
lsutil('ViewLs',mo3,li); % display the level set
fecom('ShowFiPro') % Show element properties
```

Now a volume example

```
RO=struct('dim',[10 10 40],'tolE',.1,'doSplit',1);
[model,li]=ofdemos('LS3d',RO);li{1} % Spherical cut
mo3=lsutil('cut -dosplit',model,li,RO);
cf=feplot(mo3);feplot('ShowFiMat')

% Now do a cylinder cut
li={struct('shape','cyl','xc',.5,'yc',.5,'zc',1,'nx',0,'ny',0,'nz',-1, ...
    'rc',.2,'z0',-.4,'z1',.4,'mpid',[200 300])};
mo3=lsutil('cut',model,li,RO);feplot(mo3);
cf.sel={'innode {x>=.5}','colordatamat -edgealpha.1'}
fecom('ShowFiPro') % Show element properties
```

See also

feplot

nasread

Purpose

Read results from outputs of the MSC/NASTRAN finite element code. This function is part of FEMLink.

Syntax

```
out = nasread('FileName','Command')
```

Description

`nasread` reads bulk data deck (NASTRAN input), direct reading of model and result information in OUTPUT2 and OUTPUT4 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`. Results in the `.f06` text file (no longer supported).

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 following table gives a *partial conversion list*. For an up to date table use `nas2up('convlist')`

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

`model=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 saved `OUG(i)` entries in the stack (see section 7.8). With no output argument, the result is shown in `feplot`.

Warning: do not use the `FORM = FORMATTED` in the eventual `ASSIGN OUTPUT2` statement.

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)

Translation is provided 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, ...) as curve entries in the stack (possibly multiple if various outputs are requested). Note : by default deformations are in the SDT global coordinate system (basic in NASTRAN terminology). You may switch to output in the local (global in NASTRAN terminology) using PARAM,OUGCORD,GLOBAL. To avoid *Out of Memory* errors when reading deformations, you can set use a smaller buffer sdtdef('OutOfCoreBufferSize',10) (in MB). When too large, def.def is left in the file and read as a v_handle object that lets you access deformations with standard indexing commands. Use def.def=def.def(:,:) to load all. To get the deformation in the stack use calls of the form def=stack_get(model,'curve','OUG(1)','get')
OEE	tables of element energy
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

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 the .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 (stiffness, mass, restitution 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.

Output4 text files are also supported with less performance and no support for non sequential access to data with the SDT v_handle object.

Supported options

- -full : assumes that the matrix to be read should be stored as full (default sparse).
- -transpose : transpose data while reading.
- -hdf : save data in a hdf file. Reading is performed using buffer (sdtdef('OutOfCoreBufferSize',100) for a 100MB buffer). It is useful to overcome the 2GB limit on 32 bit Matlab: see sdthdf for details about how to build v_handle on hdf file.

.f06 output (obsolete)

ASCII reading in .f06 files is slow and often generates round-off errors. You should thus consider reading binary OUTPUT2 and OUTPUT4 files, which is now the only supported format. You may try reading matrices with nasread('FileName','matprt'), tables with nasread('F','tabpt') and real modes with

```
[vector,mdof]=nasread('filename','vectortype')
```

Supported vectors are displacement (displacement), applied load vector (oload) and grid point stress (gpstress).

See also

naswrite, FEMLink

naswrite

Purpose

Formatted ASCII output to MSC/NASTRAN bulk data deck. This function is part of FEMLink.

Syntax

```
naswrite('FileName',node,elt,pl,il)
naswrite('FileName','command', ...)
naswrite('-newFileName','command', ...)
naswrite(fid,'command', ...)
```

Description

`naswrite` appends its output to the file `FileName` or creates it, if it does not exist. Use option `-newFileName` to force deletion of an existing file. 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`, `BeginTag`, `EndTag`, and `data`. Accepted commands are

`Before`	inserts data before the `BeginTag`.
`Insert`	inserts data after the `EndTag`.
`Remove`	removes a given card. Warning this does not yet handle multiple line cards.
`Set`	used to set parameter and assign values. 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}
```

When writing automated solutions, the edits should be stored in a stack entry `info,EditBulk`.

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 writing of case information.

```
femesh('reset');
femesh('testquad4')
fid=1 % fid=fopen('FileName');
naswrite(fid,'node',FEnode)
naswrite(fid,'node',FEnode)
%fclose(fid)
```

Note that `node(:,4)` which is a group identifier in SDT, is written as the SEID in NASTRAN. This may cause problems when writing models from translated from other FEM codes. Just use `model.Node(:,4)=0` in such cases.

dmig

`DMIG` writing 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 define a job handler customized to your needs and still use the nas2up calls for portability by defining `setpref('FEMLink','NASTRANJobHandler', 'FunctionName')`.

You can then run a job using `nas2up('joball','BulkFileName.dat')`. Additional arguments can be passed to the RunNastran command by simply adding them to the joball command. For example `nas2up('joball','BulkFileName.dat',struct('RunOptions','memory=1GB'))`.

It is possible provide specific options to your job handler by storing them as a `info,NasJobOpt`entry in your `model.Stack`. `nas2up('JobOptReset')` resets the default. The calling format in various functions that use the job handling facility is then

```
model=stack_set('info','NasJobOpt',nas2up('jobopt'));
nas2up('joball','step12.dat',model);
```

`RunOpt.RunOptions` stores text options to be added to the nastran command. `RunOpt.BackWd` can be used to specify an additional relative directory for the JobCpFrom command. `RunOpt.RemoteRelDir` can be used to specify the associated input for the JobCpTo command.

`nas2up('JobCpTo', 'LocalFileName', 'RemoteRelDir')` puts (copies) files to the remote directory or to `fullfile(RemoteDir,RemoteRelativeDir)` if specified.

`nas2up('JobCpFrom', 'RemoteFileName')` fetches files. The full remote file name is given by `fullfile(RemoteDir,RemoteFileName)`. Any relative directory is ignored for the local directory.

Here is a simple script that generates a model, runs NASTRAN and reads the result

```
wd=sdtdef('tempdir');

model=demosdt('demoubeam-2mat'); cf=feplot;
model=fe_case(model,'dofload','Input', ...
  struct('DOF',[349.01;360.01;241.01;365.03],'def',[1;-1;1;1],'ID',100));
model=nas2up('JobOpt',model);
```

```
model=stack_set(model,'info','Freq',[20:2:150]);

% write bulk but do not include eigenvalue options
naswrite(['-new' fullfile(wd,'ubeam.bdf')],stack_rm(model,'info','EigOpt'))

% generate a job by editing the reference mode.dat file
fname='ubeam.dat';
edits={'Set','PARAM','POST','-2';
 'replace','include ''model.bdf''','','include ''ubeam.bdf''';
 'replace','EIGRL','',nas2up('writecard',-1,[1 0 0 30],'ijji','EIGRL')};
nas2up('editbulk','mode.dat',edits,fullfile(wd,fname));
cd(wd);type(fname)
nas2up('joball',fname,model)
cg=feplot(4);mo1=nasread('ubeam.op2');
```

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('wop4','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. You can omit specifying ByteSwap at every run by setting

```
setpref('FEMLink','OutputBinaryType','ieee-le')
```

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]

Write commands are used to 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);
st=nas2up('WriteSet',-1,2000,model,selections);

curves={'curve','Sine',fe_curve('testEval -id1 sin(t)',linspace(0,pi,10)) ; ...
        'curve','Exp.',fe_curve('testEval -id100 exp(-2*t)',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);
```

- p_solid properties are implemented somewhat differently in NASTRAN and SDT, thus for a il row giving [ProID type Coordm In Stress Isop Fctn]

  In NASTRAN 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 NASTRAN 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 NASTRAN, 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.

- MAT9 and m_elastic 3 differ by the order of shear stresses yz, zx, Gxy in SDT and xy, yz, zx in NASTRAN. The order of constitutive values is thus different, which is properly corrected in SDT 6.5.

See also

nasread, ufread, ufwrite

nor2res, nor2ss, nor2xf

Purpose

Transformations from normal mode models to other model formats.

Syntax

```
[res,po,psib,cpsi] = nor2res( ... )     % sdtweb('res') for format
               RES = nor2res( ... )
         [a,b,c,d] = nor2ss ( ... )     % sdtweb('ss') for format
               SYS = nor2ss ( ... )
                xf = nor2xf ( ... )     % sdtweb('xf') for format
               ... = nor2.. (DEF,MODEL, ... )      % high level input
               ... = nor2.. (DEF,ga,MODEL, ... )
               ... = nor2.. (ga,om,pb,cp, ... )    % low level input
         ... = nor2ss ( ... , ind,fc,OutputCmd) % frequency,truncation...
         ... = nor2xf ( ... , w,ind,fc,OutputCmd)
```

Description

These functions provide detailed access, for simple high level calls see fe2ss. 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,MODEL (where the model is assumed to contain load and sensor case entries) or low level numeric arguments om,ga,pb,cp. Additional arguments w,ind,fc,OutputCmd can or must be specified depending on the desired output. These arguments are listed below.

DEF,MODEL

The normal mode shapes are given in a DEF 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 FEMLink). See also fe2ss.

Damping can be declared in different ways

- modal damping ratio can be given in DEF.data(:,2). When this column exists other damping input is ignored. This is illustrated as variable damping below.
- damp a vector of modal damping ratio can be given as the second argument nor2ss(DEF,damp,MODEL), or as an info,DefaultZeta entry as shown in the example below.
- 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,[],model,w,'hz');
  ```

  as illustrated in section 5.3.2 .

Inputs and outputs are described by a model containing a Case (see section 4.5). 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.

Examples

Here is a sample call that compares responses for two damping levels

```
[model,def]=demosdt('demogartfe');
InDof=[4.03;55.03;2.03]; OutDof=[4 55 30]'+.03;
freq=linspace(5,70,500)';
model=fe_case(model, ...
              'DofLoad','Force',InDof, ...
              'SensDof','Sensors',OutDof);
model=stack_set(model,'info','Freq',freq, ...
    'info','DefaultZeta',.01); % Ignored when def.data(:,2) exists
nor2xf(def,model,'acc iiplot "Test" -po -reset');
```

```
% Another variation
 % define variable damping in def.data(:,2)
 def.data(def.data(:,1)<30,2)=.005; % 0.5% damping below 30 Hz
 def.data(def.data(:,1)>30,2)=.02;  % 2% damping above 30 Hz
 % Truncate to first 10 modes (static correction is lost)
 d1=fe_def('subdef',def,1:12);
 % Define inputs and ouputs using DOFs (less general than fe_case)
 nor2xf(d1,InDof,OutDof,freq,'acc iiplot "Variable damping"');
iicom('ch2');ci=iiplot;ci.Stack
```

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 = demosdt('demo ubeam');
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,'Fsurf','Surface load',data)
%Sensors
model=fe_case(model,'sensdof','Sensors',[50:54]'+.03);

fe_case(model,'info')
model=stack_set(model,'info','Freq',linspace(10,240,460));

nor2xf(def,0.01,model,'iiplot "Test" -po -reset');
```

Example of transmissibility prediction using the large mass method where one defines a rigid base and a large mass such that one has 6 rigid body modes and fixed interface modes

```
model = demosdt('demo ubeam');

% define rigid base
i1=feutil('findnode z==0',model);
model = fe_case(model,'reset', ...
 'rigid append','Base',[i1(1);123456;i1(2:end)]);
% Add large mass on the base
model.Elt(end+[1:2],1:7)=[Inf abs('mass1') 0;
  i1(1) [1 1 1 1 1 1]*1e6];
```

```
def=fe_eig(model,[5 20 1e3]);  % This can be computed elsewhere

% Transmissibility for unit acceleration along x
model=fe_case(model,'DofSet','IN', ...
 struct('def',[1;0;0;0;0;0],'DOF',i1(1)+[1:6]'/100), ...
   'SensDof','OUT',[1.01;314.01]);
f=linspace(50,500,1024)';
nor2xf(def,.01,model,f,'acc iiplot "Trans-Large" -reset');

% Clean approach without the large mass
mo2=stack_set(model,'info','EigOpt',[5 14 1e3]);
mo2=fe_case(mo2,'DofSet','IN',i1(1));
SE=fe_reduc('CraigBampton -se',model);  % craig-bampton reduction
% Free modes of Craig-Bampton basis
TR=fe_eig({SE.K{:} SE.DOF});TR.DOF=SE.TR.DOF;TR.def=SE.TR.def*TR.def;

nor2xf(TR,.01,model,f,'acc iiplot "Trans-Craig"');
iicom('ch2');
```

om,ga,pb,cp

Standard low level arguments `om` (modal stiffness matrix), `ga` (modal viscous damping matrix), `pb` (modal controllability) and `cp` (modal observability) used to describe *normal mode models* are detailed in section section 5.2 . A typical call using this format would be

```
[model,def]=demosdt('demogartfe');
b = fe_c(def.DOF,[4.03;55.03])'; c = fe_c(def.DOF,[1 30 40]'+.03);
IIw=linspace(5,70,500)';
nor2xf(def.data,0.01,def.def'*b,c*def.def,IIw*2*pi, ...
  'Hz iiplot "Simul" -po -reset');
```

w,ind,fc,OutputCmd

Other arguments are

- `w` frequencies (**in rad/s** unless `Hz` is specified in `OutputCmd`) where the FRF should be computed (for `nor2xf`). Can also be given as a `model.Stack{'info','Freq'}` entry.
- `ind` (optional) gives the indices of modes to be retained (truncated modes are then added to the static correction).

- `fc` (optional) roll-off frequency : that is frequency assigned to the static correction poles. Since static correction is meant for low frequency behavior, its dynamics must be above the bandwidth of interest but where exactly can be tuned. This applies only to load input cases and a static correction must exist.

- `OutputCmd` (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 curve structure.
  `'iiplot "StackName" -po -reset'` can be used to display results in `iiplot`(see section 2.1.2). The optional `-po` is used to save poles in `ci.Stack'IdMain'` so that they can be displayed. `-reset` reinitializes the curve stack.
  `-zoh Ts` or `-foh Ts` can be used to obtained a discrete state-space model based on zero or first order hold approximations with the specified time step.

`res`

`nor2res` returns a complex mode model in the residue form

$$[\alpha(s)] = \sum_{j=1}^{2N} \frac{\{c\psi_j\}\left\{\psi_j^T b\right\}}{s-\lambda_j} = \sum_{j=1}^{2N} \frac{[R_j]}{s-\lambda_j} \tag{10.59}$$

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`).

You force use of SDT structure and rather than Control Toolbox LTI object using `setpref('SDT','UseControlToolbox',0)`. You can convert between formats using `ss_lti=nor2ss('ss2struct',ss_sdt)` or `ss_sdt=nor2ss('ss2struct',ss_lti)`.

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.

lab_in,lab_out

SDT uses fields `lab_in` and `lab_out`, while the control toolbox objects use `InputName` and `OutputName`. The commands `lab_in` are used to robust handling based on the object type.

```
lab_in =nor2ss('lab_in', sys) % Get in
lab_out=nor2ss('lab_out',sys) % Get out
sys=nor2ss('lab_in' ,sys,lab_in)  % Set in
sys=nor2ss('lab_out',sys,lab_out) % Set out
```

Theory

The basic normal mode form associated with load inputs $[b]\{u\}$ is (see section 5.2)

$$\begin{array}{c}\left[[I]\,s^2+[\Gamma]\,s+[\Omega^2]\right]_{NP\times NP}\{(s)\}=\left[\phi^T b\right]_{NP\times NA}\{u(s)\}_{NA\times 1}\\ \{y(s)\}=[c\phi]_{NS\times NP}\{p(s)\}_{NP\times 1}\end{array} \tag{10.60}$$

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

$$\begin{array}{c}\left\{\begin{array}{c}\dot{p}\,(t)\\ \ddot{p}\,(t)\end{array}\right\}=\left[\begin{array}{cc}[0] & [I]\\ -\left[\backslash\Omega^2\backslash\right] & -[\Gamma]\end{array}\right]\left\{\begin{array}{c}p\,(t)\\ \dot{p}\,(t)\end{array}\right\}+\left[\begin{array}{c}0\\ \phi^T b\end{array}\right]\{u(t)\}\\ \{y\}=[c\phi\ \ 0]\left\{\begin{array}{c}p\,(t)\\ \dot{p}\,(t)\end{array}\right\}+[0]\,\{u(t)\}\end{array} \tag{10.61}$$

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

$$\left\{ \begin{array}{c} u_a \\ q_c \end{array} \right\} = [T_I \;\; T_C] \{p\} \tag{10.62}$$

and solves the fixed interface eigenvalue problem

$$\left[T_C^T \Omega T_C - \omega_{jC}^2 T_C^T I T_C \right] \{\phi_{jC}\} = \{0\} \tag{10.63}$$

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

$$\begin{aligned} \left\{ \begin{array}{c} \dot{u} \\ \dot{q}_C \\ \ddot{u} \\ \ddot{q}_C \end{array} \right\} &= \left[\begin{array}{cc} [0] & [I] \\ \left[\begin{array}{c} 0 \\ -\hat{T}_C^T \Omega \left[T_I \;\; \hat{T}_C \right] \end{array} \right] & \left[\begin{array}{c} 0 \\ -\hat{T}_C^T \Gamma \left[T_I \;\; \hat{T}_C \right] \end{array} \right] \end{array} \right] \left\{ \begin{array}{c} u \\ q_C \\ \dot{u} \\ \dot{q}_C \end{array} \right\} + \\ & \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 0 & I \\ \hat{T}_C^T b & \hat{T}_C^T T_I \end{array} \right] \left\{ \begin{array}{c} u_F \\ \ddot{u}_a \end{array} \right\} \\ \{y\} &= \left[c T_I \;\; c \hat{T}_C \;\; 0 \;\; 0 \right] \left\{ \begin{array}{c} u_a \\ q_C \\ \dot{u}_a \\ \dot{q}_C \end{array} \right\} + [0] \left\{ \begin{array}{c} u_F \\ \ddot{u}_a \end{array} \right\} \end{aligned} \tag{10.64}$$

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

of2vtk

Purpose

Export model and deformations to VTK format for visualization purposes.

Syntax

```
opfem2VTK(FileName,model)
opfem2VTK(FileName,model,val1,...,valn)
```

Description

Simple function to write the mesh corresponding to the structure model and associated data currently in the "Legacy VTK file format" for visualization.

To visualize the mesh using VTK files you may use ParaView which is freely available at http://www.paraview.org or any other visualization software supporting VTK file formats.

```
try;tname=nas2up('tempname.vtk');catch;tname=[tempname '.vtk'];end
model=femesh('testquad4');

NodeData1.Name='NodeData1';NodeData1.Data=[1 ; 2 ; 3 ; 4];
NodeData2.Name='NodeData2';NodeData2.Data=[0 0 1;0 0 2;0 0 3;0 0 4];
of2vtk('fic1',model,NodeData1,NodeData2);

EltData1.Name ='EltData1' ;EltData1.Data =[ 1 ];
EltData2.Name ='EltData2' ;EltData2.Data =[ 1 2 3];
of2vtk('fic2',model,EltData1,EltData2);

def.def = [0 0 1 0 0 0 0 0 2 0 0 0 0 0 3 0 0 0 0 0 4 0 0 0 ]'*[1 2];
def.DOF=reshape(repmat((1:4),6,1)+repmat((1:6)'/100,1,4),[],1)
def.lab={'NodeData3','NodeData4'};
of2vtk('fic3',model,def);

EltData3.EltId=[1];EltData3.data=[1];EltData3.lab={'EltData3'};
EltData4.EltId=[2];EltData4.data=[2];EltData4.lab={'EltData4'};
of2vtk('fic4',model,EltData3,EltData4);
```

The default extention .vtk is added if no extention is given.

Input arguments are the following:

`FileName`

file name for the VTK output, no extension must be given in FileName, "FileName.vtk" is automatically created.

`model`

a structure defining the model. It must contain at least fields `.Node` and `.Elt`.
FileName and model fields are mandatory.

`vali`

To create a VTK file defining the mesh and some data at nodes/elements (scalars, vectors) you want to visualize, you must specify as many inputs *vali* as needed. *vali* is a structure defining the data: $vali = struct('Name', ValueName, 'Data', Values)$. Values can be either a table of scalars ($Nnode \times 1$ or $Nelt \times 1$) or vectors ($Nnode \times 3$ or $Nelt \times 3$) at nodes/elements. Note that a deformed model can be visualized by providing nodal displacements as data (e.g. in ParaView using the "warp" function).

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 within each library you may want to specify options (direct, iterative, in-core, out-of-core, parallel, ...).

Using the `ofact` object in your code, lets you specify method at run time rather than when writing the code. Typical steps are

```
ofact('method spfmex'); % choose method
kd = ofact(k);          % create object and factor
static = kd\b           % solve
ofact('clear',kd)       % clear factor when done
```

For single solves `static=ofact(k,b)` performs the three steps (factor, solve clear) in a single pass.

The first step of method selection provides an open architecture that lets users introduce new solvers with no need to rewrite functions that use `ofact` objects. Currently available methods are listed simply by typing

```
>> ofact

Available factorization methods for OFACT object
->  spfmex : SDT sparse LDLt solver
   sp_util : SDT skyline solver
        lu : MATLAB sparse LU solver
    mtaucs : TAUCS sparse solver
   pardiso : PARDISO sparse solver
      chol : MATLAB sparse Cholesky solver
   *psldlt : SGI sparse solver (NOT AVAILABLE ON THIS MACHINE)
```

and the method used can be selected with `ofact('method MethodName')`. SDTools maintains pointers to pre-compiled solvers at `http://www.sdtools.com/faq/FE_ofact.html`.

The factorization `kd = ofact(k);` and resolution steps `static = kd\b` can be separated to allow multiple solves with a single factor. Multiple solves are essential for eigenvalue and quasi-newton solvers. `static = ofact(k)\b` is of course also correct.

The clearing step is needed when the factors are not stored as MATLAB variables. They can be stored in another memory pile, in an out-of-core file, or on another computer/processor. Since for large problems, factors require a lot of memory. Clearing them is an important step.

Historically the object was called `skyline`. For backward compatibility reasons, a `skyline` function is provided.

umfpack

To use UMFPACK as a `ofact` solver you need to install it on your machine. This code is available at www.cise.ufl.edu/research/sparse/umfpack.

pardiso

Based on the Intel MKL (Math Kernel Library), you should use version 8 and after.

By default the pardiso call used in the ofact object is set for symmetric matrices. For non-symmetric matrices, you have to complement the ofact standard command for factorization with the character string `'nonsym'`. Moreover, when you pass a matrix from Matlab to PARDISO, you **must transpose** it in order to respect the PARDISO sparse matrix format.
Assuming that k is a real non-symmetric matrix and b a real vector, the solution q of the system $k.q = b$ is computed by the following sequence of commands:

```
ofact pardiso                    % select PARDISO solver
kd = ofact('fact nonsym',k');    % factorization
q=kd\b;                          % solve
ofact('clear',kd);               % clear ofact object
```

The factorization is composed of two steps: symbolic and numerical factorization. For the first step the solver needs only the sparse matrix structure (i.e. non-zeros location), whereas the actual data stored in the matrix are required in the second step only. Consequently, for a problem with a unique factorization, you can group the steps. This is done with the standard command *ofact('fact',...)*.
In case of multiple factorizations with a set of matrices having the same sparse structure, only the second step should be executed for each factorization, the first one is called just for the first factorization. This is possible using the commands `'symbfact'` and `'numfact'` instead of 'fact' as follows:

```
kd = ofact('symbfact',k);    % just one call at the beginning
...
kd = ofact('numfact',k,kd); % at each factorization
q=kd\b;                      %
...
ofact('clear',kd);           % just one call at the end
```

You can extend this to **non-symmetric systems** as described above.

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 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).

Generic commands

verbose

Persistent solver verbosity handling. By default, solvers tend to provide several information for debugging purposes. For production such level of verbosity can be undesirable as it will tend to fill-up logs and slow down the process due to multiple display outputs. One can then toggle the `silent` option of `ofact` with this command.

`ofact('silent','on');`, or `ofact('silent')` will make the solver silent. `ofact('silent','off');` will switch back the solver to verbose.

It is possible to activate the verbosity level during the solver selection, using token `-silent` to get a silent behavior or `-v` to get a verbose behavior. **Note that a space must exist between the solver name and other tokens**.

```
ofact('spfmex -silent') % selected the spfmex_utils solver with silent option
ofact('spfmex -v') % selects the spfmex_utils solver with verbose option
```

_sel

Advanced solver selection with parameter customization. Solvers use default parameters to work, but it is sometimes usefull to tweak these values for specific configurations. This command further allows generic solver selection from GUI inputs.

By default, one can call `ofact('_sel','solver')`, possibly with the `-silent` token. Direct parameter tweaking is currently supported for `spfmex` only, where the `MaxDomainSize` (default to 32), and `MaxZeros` (default to 0.01) can be provided. For larger models, it is suggested to use a `MaxZeros` value set to 0.1.

```
ofact('_sel','spfmex 32 .1') % tweaks the MaxZeros spfmex solver value to 0.1
```

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 another computer on the network.

`silent`

`silent` handled the verbosity level of your solver.

See also `fe_eig`, `fe_reduc`

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 perm2sdtfunction reads PERMAS model, result and matrices files. Binary and ASCII files are supported.

filesModel 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, FDPIPE2, 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

perm2sdtreads binary and ASCII .mtl file format. The syntax is

perm2sdt('binary.mtl File.mtl') for binary files and perm2sdt('ascii.mtl File.mtl') for ASCII files.

See also

FEMLink

polytec

Purpose

Reading of POLYTEC .svd files.

Syntax

```
wire         = polytec('ReadMesh',fname);
list         = polytec('ReadList',fname)
XF           = polytec('ReadSignal',fname,RO)
[cmap,fname] = polytec('ReadImg',fname);
polytec('ToMat',fname,RO);
```

Description

The polytecfunction reads files generated by polytec measurement systems. Actual files successfully read are :

- **.set* : Setting file from which the wireframe can be loaded if defined (useful for pretest analysis before measurement)
- **.pvd* : File containing a unique channel from which measurements can be extracted
- **.svd* : File containing several channels from which the wireframe geometry and the data can be extracted depending what measurements have been performed (Time/Frequency domain, Transfers...)

Prior to use this function, the *Polytec File Access* provided by *Polytec* must be installed : Download the *Polytec Update* software (freely available in their website) and install it with all the dependencies.

This function has been tested only with a few versions of *Polytec File Access* (4.7, 5.0 and 5.6) , but we experienced no problem at each update so that it is likely to work with all versions in between.

It is possible for some application to merge several measurements files with the *Polytec* applications. This is handled by this function : simply provide the file defining the merge (and eventually the individual files if links are made and not data copy).

ReadMesh

The ReadMesh command allows to extract the test wireframe in the SDT format. It contains the node locations and the sensor orientions (depending on the type of laser used : monopoint, with mirror, 3D laser)

```
fname=sdtcheck('patchget',struct('fname','PolytecMeas.svd'));
wire=polytec('ReadMesh',fname); % Set the wireframe in the variable wire
polytec('ReadMesh',fname); % Without output, directly open the model in feplot
```

ReadList

Three parameters are needed to access the data :

- `pointdomain` : Time, FFT, 1/3 octave...
- `channel` : Vib, Ref1, Vib & Ref1 (transfers)...
- `signal` : Displacement, Velocity, Acceleration, H1 Displacement / Force...

The `ReadList` command allows to see all the combinations of these parameters that are allowed for a given file. With an output, the call sends back a cell array containg all the cominations. Without, it opens a tree in a tab in the `SDTRoot` window, from which it is posibble to do a *right click* on the wanted data and select *Read Selected* : data are read and display in an `iiplot` window.

```
fname=sdtcheck('patchget',struct('fname','PolytecMeas.svd'));
% Provide a cell array with all readable measured data
list=polytec('ReadList',fname);
display(list);
polytec('ReadList',fname); % Open a tree in SDTRoot to interactively select data
```

ReadSignal

The `ReadSignal` command allows to read the measurement data specified by the three parameters (*point domain, channel* and *signal*) given in a structure as a third argument.

It is also possible to provide the wanted parameters by extracting the corresponding line of the cell array provided by the command `ReadList` (as a parameter *list*).

```
fname=sdtcheck('patchget',struct('fname','PolytecMeas.svd'));
RO=struct('pointdomain','FFT','channel','Vib & Ref1',...
'signal','H1 Displacement / Voltage');
XF=polytec('ReadSignal',fname,RO);
% alternative call using one row of the cell array "list"
list=polytec('ReadList',fname);
XF=polytec('ReadSignal',fname,struct('list',{list(20,:)}));
```

ReadImg

The ReadImg command allows to read the image used to construct the test geometry. The image is displayed in a figure, the RGB colormap is provided as first output and it creates a *.png* file whose name is given as second output.

```
fname=sdtcheck('patchget',struct('fname','PolytecMeas.svd'));
[cmap,fname] = polytec('ReadImg',fname);
```

ToMat

The ToMat command allows to extract wanted data and save them as a *.mat* file in the SDT format. This is useful especially if the scripts that read the *Polytec* files must be run on a *Linux* OS or on computers where *Polytec File Access* is not installed.

Once the ToMat file has been executed, the *.mat* file is used insted of the *Polytec* one to load data (the ReadMesh and ReadSignal commands remain the same).

```
fname=sdtcheck('patchget',struct('fname','PolytecMeas.svd'));
RO=struct('pointdomain','FFT','channel','Vib & Ref1',...
'signal','H1 Displacement / Voltage');
% Create a .mat file next to the Polytec one with the mesh, the data and the image.
polytec('ToMat',fname,RO);
r1=load(strrep(fname,'.svd','.mat'));
r1.TEST
r1.XF
r1.img
```

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. [12], 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 $\psi_{PT} = \psi_T + \Delta\psi$ verifies the properness condition $\psi_{PT}\psi_{PT}^T = 0$. 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( ... ,'iiplot ...')
```

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

$$\mathtt{xf} = [C]\,(s\left[\backslash I_\backslash\right] - [A])^{-1}\,[B] + [D] \tag{10.65}$$

The result is stored in the xf format (see details page 214). 'iiplot "Test" -po -reset' can be used to display results in iiplot(see section 2.1.2). The option -po is used to save poles in ci.Stack{'IdMain'} so that they can be displayed. -reset reinitializes the curve stack.

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 214), qbode computes the FRF at the frequency points w

$$\mathtt{xf} = \frac{\mathtt{num}(j\omega)}{\mathtt{den}(j\omega)} \tag{10.66}$$

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

$$\frac{[R]}{s-\lambda}+\frac{[\bar{R}]}{s-\bar{\lambda}}=2\frac{(s\mathrm{Re}(R))+(\zeta\omega\mathrm{Re}(R)-\omega\sqrt{1-\zeta^2}\mathrm{Im}(R))}{s^2+2\zeta\omega s+\omega^2} \tag{10.67}$$

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 202) 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 5.6) and the state space forms (see section 5.4). You can use either high level calls with data structures or low level calls providing each argument

```
ci=demosdt('demo gartid est')
SYS = res2ss(ci.Stack{'IdMain'});
RES = ss2res(SYS);
ID=ci.Stack{'IdMain'};
[a,b,c,d] = res2ss(ID.res,ID.po,ID.idopt);
```

Important properties and limitations of these transformations are

`ss`

- 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 2.8.1). To obtain models with multiple poles use `id_rm` to generate `new_res` and `new_po` matrices.
- you can bypass the `id_rm` call by providing complex mode modal controllability $\psi_j^T b$ in a `.psib` field and modal observability $c\psi_j$ in a `.def` field. This is in particular used by `fe2ss` with the `-cpx` command option.
- `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.

`res`

- 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 2.8.1 , 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

$$\begin{Bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{Bmatrix} = \begin{bmatrix} [0] & \begin{bmatrix} \backslash I \backslash \end{bmatrix} \\ -\begin{bmatrix} \backslash \omega_j^2 \backslash \end{bmatrix} & -\begin{bmatrix} \backslash 2\zeta_j\omega_j \backslash \end{bmatrix} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} + \begin{Bmatrix} B_1 \\ B_2 \end{Bmatrix} \{u(t)\}$$
$$\{y(t)\} = [C_1 \;\; C_2] \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} \tag{10.68}$$

where the blocks of matrices B_1, B_2, C_1, C_2 are given by

$$\begin{Bmatrix} C_{1j} \\ C_{2j} \end{Bmatrix} = [\mathrm{Re}\,(c\psi_j) \quad \mathrm{Im}\,(c\psi_j)] \frac{1}{\omega_j\sqrt{1-\zeta_j^2}} \begin{bmatrix} \omega_j\sqrt{1-\zeta_j^2} & 0 \\ \zeta_j\omega_j & 1 \end{bmatrix}$$
$$\begin{Bmatrix} B_{j1} \\ B_{j2} \end{Bmatrix} = 2 \begin{bmatrix} 1 & 0 \\ -\zeta_j\omega_j & -\omega_j\sqrt{1-\zeta_j^2} \end{bmatrix} \begin{bmatrix} \mathrm{Re}\left(\psi_j^T b\right) \\ \mathrm{Im}\left(\psi_j^T b\right) \end{bmatrix} \tag{10.69}$$

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 212), `res2tf` generates the corresponding polynomial transfer function representation (see `tf` page 214)).

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 = feval(id_rc('@rms'),t,w)
rm = feval(id_rc('@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

$$\texttt{rms} = \sqrt{\frac{1}{2\pi}\int_{\omega_1}^{\omega_2} |t(\omega)|^2 d\omega} = \sqrt{\frac{1}{2\pi}\int_{\omega_1}^{\omega_2} a(\omega) d\omega} \tag{10.70}$$

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`

samcef

Purpose

Interface function with SAMCEF FEM code.

Syntax

```
Up=samcef('read model.u18')
Up=samcef('read model.u18','buildup')
Up=samcef('read model.bdf','buildup')
a=samcef('lectmat','FileRoot')
samcef('write FileName',model)
```

Description

read

The `read` command import : models from `.dat files`, results from `.u18` file. With the `'buildup'` argument, the `.u11` and `.u12` files are also read to import element matrices into a superelement. Additional DOFs linked to reduced shear formulations are properly condensed. Note that to export a standardized form of the model, you should use the `.SAUV BANQUE "FileName.dat"` command in SAMCEF.

When reading a `.u18` file, it may be necessary to import the properties from the model to clarify which DOFs are actually used in the model. You should thus have a `.data` file with the same root name in the same directory. Modeshapes are stored in the model stack entry `curve,record(12)_disp`. Other imported results are also stored in the stack.

write

Basic writing is supported with `samcef('write FileName',model)`. Please send requests to extend these capabilities.

conv

This command lists conversion tables for elements, topologies, facetopologies. You can redefine (enhance) these tables by setting preferences of the form `setpref( 'FEMLink', 'samcef.list', value)`, but please also request enhancements so that the quality of our translators is improved.

See also

FEMLink

sd_pref

Purpose

Safe MATLAB preferences handling.

Syntax

```
sd_pref('set','Group','Pref','val');  % setpref
flag=sd_pref('get','Group','Pref');   % getpref
i1=sd_pref('is','Group');             % ispref
sd_pref('rm','Group','Pref');         % rmpref
```

Description

MATLAB, and MCR, have known issues of preference file corruption if accessed by several instances at once. To avoid this issue `sd_pref` implements a safe access strategy using `fjlock` .

The syntax is equivalent to ususal MATLAB `*pref` commands, the additional first argument provides the function prefix to be used.

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+^>')
```

sdtcheck

Purpose

Installation handling and troubleshooting.

Description

For SDT to run in MATLAB the path to SDT functions must be added to the MATLAB search path. Additional libraries are also required that sometimes need an explicit declaration in MATLAB. `sdtcheck` then packages manual input to alter the user MATLAB settings if needed.

Commands

`path`

This command properly defines the MATLAB search path to run SDT. It has to be used at startup if the search path was not saved in your MATLAB session with SDT installed.

```
% Initialization of SDT in MATLAB path
pw0=pwd;
cd('path_to_my_sdt')
sdtcheck path
cd(pw0)
```

`patchJavaPath[,set]`

SDT GUI utilities are based on Java and require additional Java libraries to be loaded by MATLAB. To ensure proper SDT GUI running the user needs to alter the default MATLAB `classpath.txt`.

- Command `patchJavaPath` checks whether the Java `classpath` contains the libraries needed by SDT. If not a warning will be issued along with an executable link to modify the Java `classpath`.
- Command `patchJavaPathSet` generates a custom Java `classpath` for the user MATLAB configuration to add the libraries required by SDT. Note that you will need to restart MATLAB for the modification to be effective.

This setup is highly depending on the MATLAB version

- For MATLAB versions greater or equal to `8.0` (from `R2012b`). There is no known issue for the Java path setup.

- For MATLAB versions up to 7.14 (up to R2012a). The setup strategy allows local customization but using a file that will impact all MATLAB versions. By default this function thus attemps to alter the base MATLAB file. For certain users, this operation can be not permitted, and it is then advised to run the following command

  ```
  sdtcheck('PatchJavaPathSet-forceStartupDir')
  ```

  Be aware that this command will add a file in your startup directory. Make sure to delete it before launching other MATLAB versions from the same startup directory. Corruption and failed Desktop launches can otherwise occur.

patchFile[,set]

To distribute more intricate examples, SDTools uses patches in the form of zip files downloaded to the `fullfile(sdtdef('tempdir'),'sdtdemos')` directory and possibly extracted in the same directory. For example

```
fname=sdtcheck('PatchFile',struct('fname','DbTest.unv','in','DrumBrake.zip'));
```

will search a `DbTest.unv` file and if not found will download the `DrumBrake.zip` set of files where `DbTest.unv` is expected to be located.

patchMkl[,path,_rt]

The new ofact solver based on MKL Pardiso requires additional libraries to run. patchMkl packages its installation.

- patchMkl downloads and installs the libraries.
- patchMklPath verifies the search path and library path.
- patchMkl_rt provides troubleshooting information regarding library installation.

SdtRootDir

Provides the SDT root directory.

```
wd=sdtcheck('SdtRootDir')
```

sdtdef

Purpose

Internal function used to handle default definitions.

Syntax

```
sdtdef('info')
[i1,r1]=sdtdef('in','Pref')
sdtdef('Pref')
sdtdef('Group.Pref')
sdtdef('Pref',Value)
sdtdef('Pref-safe',Value)
sdtdef('Pref-SetPref',Value)
```

Description

Allows to handle preferences of SDT, FEMLink and OpenFEM.

This function was initially developped to limit the risks of curruption of the MATLAB preference file, which can occur if multiple instances of MATLAB try to access this file at the same time with standard commands getpref/setpref.

To handle preferences of SDT, FEMLink and OpenFEM, the recommended use is to

- setpref('[SDT,OpenFEM,FEMLink]','Pref','value') for the first creation of the preference.
- [i1,r1]=sdtdef('in','Pref') to check if a preference is defined and get back the value .
- sdtdef('[,OpenFEM.,FEMLink.]Pref','value') to perform a local modification of the preference value (in the current MATLAB session).
- sdtdef('[,OpenFEM.,FEMLink.]Pref-safe','value') only perfoms the local modification if the preference does not exist (previous call fail in this case)
- sdtdef('[,OpenFEM.,FEMLink.]Pref-SetPref','value') performs a hard modification of the preference (through a setpref). Only works if the preference already exists, only setpref can be used to create a preference for the first time.

To reset values to factory defaults use sdtdef('factory').

info

The command `sdtdef('info')` provides the full list of preferences of `SDT`.

The command `sdtdef('info','OpenFEM')` provides the full list of preferences of `OpenFEM`.

The command `sdtdef('info','FEMLink')` provides the full list of preferences of `FEMLink`.

The command `sdtdef('info','SDTools')` provides the full list of preferences of `SDTools`.

in

To check if a preference already exists in order to create it with `setpref` if not, use `[i1,r1]=sdtdef('in','[,OpenFEM.,FEMLink.]Pref')`. It tells if `'Group','Pref'` exists in i1 as bool, and provides value r1 if true, empty if false.

With an empty `Pref`, the full list of preferences in the `Group` is forwarded in `r1` if the group exists.

SDT preferences

Preferences of SDT are accessed directly by the call `sdtdef('Pref')` (replaced by the standard call `getpref('SDT','value')`. It returns an error if the preference does not exist.

Here is a partial list of SDT preferences :

- `avi` : cell array of default AVI properties, see the MATLAB `avifile` command.
- `DefaultZeta` :Default value for the viscous damping ratio. The nominal value is `1e-2`. The value can also be specified in a model stack and is then handled by `fe_def defzeta` and `fe_def defeta` commands.
- `KikeMemSize` : Memory in megabytes used to switch to an out-of-core saving of element matrix dictionaries.
- `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, ...
- `tempdir` : can be used to specify a directory different than the `tempdir` returned by MATLAB. This is typically used to specify a faster local disk.

- OutOfCoreBufferSize : Memory in bytes used to decide switching to an out-of-core procedure. This is currently used by nasread when reading large OUTPUT2 files.

FEMLink preferences

Preferences of SDT are accessed directly by the call sdtdef('FEMLink.Pref') (replaced by the standard call getpref('FEMLink','value'). It returns an error if the preference does not exist.

Here is a partial list of FEMLink preferences :

- CopyFcn : command used to copy file to remote locations. See naswrite job commands.
- DmapDir : directory where FEMLink is supposed to look for NASTRAN DMAP and standard files.
- NASTRAN : NASTRAN version. This is used to implement version dependent writing of NASTRAN files.
- RemoteDir : location of remote directory where files can be copied. See naswrite job commands.
- SoftwareDocRoot : defines the path or URL for a given software. You can use sdtweb('$Software/file.html') commands to access the proper documentation. For example

```
setpref('FEMLink','SdtDocRoot', ...
'http://www.sdtools.com/help/');
sdtweb('$sdt/sdt.html');
```

- TextUnix : set to 1 if text needs to be converted to UNIX (rather than DOS) mode before any transfer to another machine.

OpenFEM preferences

Preferences of SDT are accessed directly by the call sdtdef('OpenFEM.Pref') (replaced by the standard call getpref('OpenFEM','value'). It returns an error if the preference does not exist.

Here is a partial list of OpenFEM preferences :

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	stack pointer (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 and a field of vfields is accessible using GetData. To get the model of a cf FeplotFig, you may use the syntax cf.mdl.GetData.

See also

feplot, idopt, iiplot, ii_mac, xfopt

sdthdf

Purpose

Description

sdthdf handles MATLAB data/metadata information. Its main purpose if to deal efficiently with the binary MATLAB file format .mat that is based on the HDF file format.

The new hdf5 file format, supported by MATLAB since version 7.3, allows very efficient data access from files. Partial loading is possible, as well as data location by pointers. sdthdf allows the user to unload RAM by saving specific data to dedicated files, and to optimize file loading using pointers. To be able to use these functionalities, the file must have been saved in hdf5 format, which is activated in MATLAB using the -v7.3 option of the save function.

File handling commands based on HDF5

The following commands are supported.

hdfReadRef

This command handles partial data loading, depending on the level specified by the user.

For unloaded data, a v_handle pointer respecting the data structure and names is generated, so that the access is preserved. Further hdfreadref application to this specific data can be done later.

By default, the full file is loaded. Command option -level allows specifying the desired loading level. For structured data, layers are organized in which substructures are leveled. This command allows data loading until a given layer. Most common levels used are given in the following list

- -level0 Load only the data structure using pointers.
- -level1 Load the data structure and fully load fields not contained in substructures.
- -level2 Load the data structure, and fully load fields including the ones contained in the main data substructures
- -level100 Load the data structure, and fully load all fields (Until level 100, which is generally sufficient).

It takes in argument either a file, or a data structure containing hdf5 v_handle pointers. In the case where a file is specified, the user can precise the data to be loaded, by giving its named preceded by a slash /, substructure names can also be specified giving the name path to the variable to be loaded with a succession of slashes.

```
% To load an hdf5 file
r1=sdthdf('hdfreadref','my_file.mat');
% To load it using \vhandle pointers
r1=sdthdf('hdfreadref-level0','my_file.mat');
% To load a specified variable
r2=sdthdf('hdfreadref-level0','my_file.mat','/var2');
% To load a specified sub data
r3=sdthdf('hdfreadref-level1','my_file.mat','/var2/subvar1');
% To load a subdata from a previously loaded pointer
r4=sdthdf('hdfreadref',r2.subvar1);
```

hdfdbsave

This command handles partial data saving to a temporary file. It is designed to unload large numerical data, such as sparse matrices, or deformation fields. Command option `-struct` however allows to save more complex data structures.

The function takes in argument the data to save and a structure with a field `Dbfile` containing the temporary file path (string). The function outputs the `v_handle`to the saved data. The `v_handle`has the same data structure than the original. The `v_handle`data can be recovered by `hdfreadref`.

```
opt.Dbfile=nas2up('tempname_DB.mat');
r1=sdthdf('hdfdbsave',r1,opt);
r2=sdthdf('hdfdbsave-struct',r2,opt);
```

hdfmodelsave

This command handles similar saving strategy than `hdfdbsave` but is designed to integrate `feplot`models in `hdf5` format. The file linked to the model is not supposed to be temporary, and data names are linked to an SDT model data structure, which are typically in the model stack. The variable data names, must be of format `field_name` to store `model.field` in `hdf5` format.

For model stack entries, the name must be of the type `Stack_type_name` to store `cf.Stack{'type','name'}`.

The function takes in argument the data base file, the feplot handle and the data name, which will be interpreted to be found in the `feplot`model. The data will be replaced by `v_handle`pointers in the `feplot`model. Data can be reloaded with command `hdfmodel`

```
sdthdf('hdfmodelsave','my_file.mat',cf,'Stack_type_name');
```

hdfmodel

This command loads v_handledata pointers in the feplotmodel at locations where hdf5 data have been saved. This command works from the hdf file side, and loads all the data contained with standard names in the feplotmodel. See hdfmodelsave for more information on the standard data names. Commando option -check only loads the data contained in the hdf file that is already instanced in the feplotmodel.

```
sdthdf('hdfmodel','my_file.mat',cf);
```

hdfclose

Handling hdf5 files in data structures can become very complex when multiple handles are generated in multiple data. This command thus aims to force a file to be closed.

```
sdthdf('hdfclose','my_file.mat');
```

A lower level closing call allows clearing the hdf5 libraries, when needed,

```
sdthdf('hdfH5close')
```

Here is an example of offload to HDF5 based mat files, and how to access the data afterwards.

```
fname=fullfile(sdtdef('tempdir'),'ubeam_Stack_SE.mat');
fname2=fullfile(sdtdef('tempdir'),'ubeam_model.mat');
model=demosdt('demoubeam');cf=feplot;
cf.mdl=fe_case(cf.mdl,'assemble -matdes 2 1 NoT -SE');
cf.Stack{'curve','defR'}=fe_eig(cf.mdl,[5 50 1e3]);

% save(off-load) some stack entries to a file
sdthdf('hdfmodelsave',fname,cf,'Stack_curve_defR')
% save model but not the off-loaded entries
fecom('save',fname2);

cf=fecom('load',fname2); % reload the model
sdthdf('hdfmodel',fname,cf); % reload pointers to the entries
cf.Stack{'defR'}
```

For MATLAB ¿7.3 HDF based .mat files, you can open a v_handle pointer to a variable in the file using

```
fname=fullfile(sdtdef('tempdir'),'ubeam_Stack_SE.mat');
var=sdthdf('hdfreadref -level0',fname,'Stack_curve_defR')
```

`ioClearCache,ioLoad, ...`

`io` commands are meant to allow I/O operations tailored to memory demanding operations.

`sdthdf('ioFreeCache','fname')` or `sdthdf('ioFreeCache','_vhandlename')` free the cache of a given file or the file associated with a specific `v_handle`.

`sdthdf('ioLoadVarName','fname')` loads `VarName` from file `fname` and frees the associated cache. This operation still requires memory to store the variable and the file cache and may thus fail for large variables.

`sdthdf('ioBufReadVarName','fname')` will load `VarName` from file `fname` while controlling the cache used. This is only intended for large data sets written to file as contiguous uncompressed data.

MATLAB data handling utilities

`compare`

The `compare` command checks the data equivalence of two MATLAB variables. This is an efficient utility to spot local differences in large or complex data.

Any data compound can be input, mixing any native MATLAB classes. The `compare` command will then recursively check the equivalence of the data compound structure and content.Its output will be a cell array with as many lines as differences were found. The cell array output is empty if all fields were found equal.

```
% Comparing two sets of data compounds
r1=struct('data1',{{speye(15)}},'data2',rand(15,1));
r2=struct('data1',{{speye(14)}},'data2',rand(15,1),...
'data3',1);
sdthdf('compare',r1,r2)
```

`pointerList[sortm,-mb]`

The `pointerList` command outputs the internal memory address of each variable, (expanded for structures and cell arrays) specified in input and provides a statistic on the total amount of data pointed in memory versus the total memory allocated to the storage. As MATLAB performs lazy variable copy, copied variables share the same pointed memory data until one of the instances is modified, the traditional output of the `who` command may thus be inappropriate to assess memory usage. The following command options allow output variations

- sortm sorts the output in increasing memory, so that the user sees the largest memory usage at the bottom of the command window.
- -mb converts the memory sizes outputs from Bytes to Megabytes.

If not output is specified, the statistics are directly printed on screen, else a cell array with as many lines as found variables is output, and three columns. First column is the variable name, second is the memory address, third is the memory size.

The input is required to be a structure, cell array, v_handle object or a string containing whos. In the latter case, a reformatting of the output of the whos command is performed.

```
% Getting information on data sizes in memory
% Generate a sample data structure
r1=struct('data1',speye(12),'data2',rand(15,1));
r1.data3=r1.data1; % lazy copy

% reformat the output of whos
sdthdf('pointerlistsortm','whos')

% Get memory information on r1
sdthdf('pointerlistsortm',r1)
```

See also

SDT handle

sdtroot, sdt_locale

Purpose

Base SDT gui figure handling.

Description

This function is used to implement base SDT mechanisms for tabs shown in JAVA GUI. It also supports advanced structure manipulations (stored here due to interactions with structure like objects aka handles and other SDT objects)

Init

Commands for tab initialisation/refreshing. `InitPTree` is an example of initialization of the navigation pane. `InitProject` implements the typical project tab which is detailed in section 8.1.2 . `InitPref` opens the SDT preference editor.

Set

Commands for property setting. Implements `SetPref` for preferences, `SetProject` for generic project parameters and and generic setting of fields defined in `PARAM`.

The default mechanism for set is to specify the tab in the command and provide data to be set a structure where each field describes a cell in the tab. An example for the `Project` tab.

```
tdir=sdtdef('tempdir');
sdtroot('SetProject',struct('ProjectWd',tdir, ... % Root file location
    'PlotWd',fullfile(tdir,'plots'), ...          % Plot directory
    'PlotWord',fullfile(tdir,'tmp_word.docx')));  % Word file for image insert
```

PARAM

`PARAM` commands are used to retrieve data stored normally stored in the `userdata` of the project figure.

- `PA=sdtroot('paramVH')` gives a `v_handle` to the main project data structure.
- `RO=sdtroot('PARAM2RO')` resolves all java dependencies and returns a basic MATLAB `struct` containing all parameters.

- To access individual data prefer calls with field names. The -safe option performs inits if needed.

```
st=sdtroot('PARAM.Project.ProjectWd -safe')
r1=sdtroot('PARAM.Project')
```

- sdtroot('PARAMWord') initializes for potential export based on content of PlotWd and PlotWord

OsDic

Each project figure supports a dictionnary (or OsDic) of named comgui objSet styles that can be used to format figures, images, ... The following illustrates simple manipulations, for a list of usual categories see section 8.1 .

```
% Sample style definition, see examples in d_imw
my_style={'position',[NaN,NaN,1087,384],'@line',{'linewidth',5}};
sdtroot('InitOsDic'); % Display list of named styles
sdtroot('setOsDic',{'ImMyStyle',my_style}) % Associate ImMyStyle name to this style
figure(1);plot([0 1]);
comgui('objset',1,{'@OsDic(SDT Root)','ImMyStyle'}); % Apply named style
```

@sfield

Subcommand sfield provides structure manipulation utilities. It can be accessed by calling sfield=sdtroot('@sfield');.

The following commands are available

- AddMissing Completes a structure with fields found missing from a default structure. r1=sfield('AddMissing',r1,r2). Inputs r1 is the working structure, r2 is a default structure. Output r1 is the input structure for which fields of r2 that were not present have been added. Field names are case sensitive.

```
r1=struct('PostCheck',1,'Opt','test','PostName','NameP','run',true);
r2=struct('Opt','ttt','other','value');
r1=feval(sdtroot('@sfield'),'AddMissing',r1,r2);
```

- AddSelected Completes a structure with fields found missing from a default structure, for a given list of field names to intersect. r1=sfield('AddSelected',r1,r2,list);. Inputs r1 is the working structure, r2 is a default structure, list is a list of field names to consider. Output r1 is the input structure for which fields of r2 that were not present and intersected in list have been added. Field names are case sensitive.

```
r1=struct('PostCheck',1,'Opt','test','PostName','NameP','run',true);
r2=struct('Opt','ttt','other','value','field',true);
r1=feval(sdtroot('@sfield'),'AddSelected',r1,r2,{'field'});
```

- AddIncF Adds to a main structure the fields from another structure, fields names in the second structure that are already present are incremented with a number added to the field name end. r1=sfield('AddIncF',r1,r2).

```
r1=struct('PostCheck',1,'Opt','test','PostName','NameP','run',true);
r2=struct('Opt','ttt');
r1=feval(sdtroot('@sfield'),'AddIncF',r1,r2);
```

- Cell2Struct Robust transform of a cell array in format {tag,data,...}, or {tag,data; ...} to a structure. r1=sfield('Cell2Struct',list);.

```
list={'tag',{'value','test'},'opt',1};
r1=feval(sdtroot('@sfield'),'Cell2Struct',list);
```

- GetField Case insensitive field recovery. val=sfield('GetField',r1,field,typ); r1 is an input structure, field is the field name to recover, typ is the output wanted, if set to 'name' the fieldname is output, the associated value is provided otherwise.

```
r1=struct('PostCheck',1,'Opt','test','PostName','NameP','run',true);
v1=feval(sdtroot('@sfield'),'GetField',r1,'postname','field');
f1=feval(sdtroot('@sfield'),'GetField',r1,'postname','name');
```

- MergeI Merge two structures into a single one with case insensitive field name union. r1=sfield('MergeI',r1,r2);. Output r1 is a structure with merged fields of inputs r1 and r2 with priority given on r1.

```
r1=struct('PostCheck',1,'Opt','test','PostName','NameP','run',true);
r2=struct('opt','ttt','other','value');
r1=feval(sdtroot('@sfield'),'MergeI',r1,r2);
```

- Sub Recovers fields from a structures whose names match a given pattern (through a regular expression). r2=sfield('Sub',r1,pat,typ):. Output r2 is a structure whose fields are fields from input structure r1 that were matched with pat as a regular expression. If typ is set to true, the matched pattern is removed from the output field name, kept otherwise.

```
r1=struct('PostCheck',1,'Opt','test','PostName','NameP','run',true);
r2=feval(sdtroot('@sfield'),'Sub',r1,'^Post',1);
```

sdtweb

Purpose

SDT file navigation function.

Description

This function allows opening the SDT documentation, opening classical file types outside Matlab, and source code navigation.

OpenFileAtTag

When not called by a command starting with _, sdtweb opens a file.

The documentation can be displayed at two locations :

- In the MATLAB help browser : define this location as default with
  `sdtdef('browser-SetPref','')` or
  `sdtdef('browser-SetPref','-helpbrowser')`
- In the MATLAB web browser : : define this location as default with
  `sdtdef('browser-SetPref','hack')`

(Note that without the `-SetPref`, the displayed location is only modified for the current session, which is useful to temporarily switch from one display to the other.)

Their is a MATLAB bug when displayed in the help browser : links to locations on a page sometimes do not work properly, so that using the web browser is more convienient for now. It is recommended to use the help browser only to do a research in the documentation or if the table of content is really needed.

The main cases are

```
sdtweb feutil              % Html documentation of feutil
sdtweb feutil#Renumber     % at a tag in the HTML file
sdtweb feutil#Renumber -browser  % same but in external browser
sdtweb feutil('renumber') % open .m file at tag 'renumber'
sdtweb source.c#tag        % source.c file at tag
sdtweb file.doc            % opens word for a given file.doc
```

`sdtweb('_path')` lists the help search path. `sdtweb('_pathReset')` redefines preferences.

Utils

`sdtweb('_link','callback','comment')` creates a clickable link.

`sdtweb('_links','callback','comment')` creates a clickable link showing just the comment.

`sdtweb('_wd',wd0,wd1)` recursively searches for a subdirectory of `wd0` named `wd1`. Command option `-reset` regenerates the underlying directory scan.

`sdtweb('_fname',fname,wd0)` recursively searches for a file named `fname` in `wd0` or any of its subdirectories, or the current directory.

`sdtweb('_find','base_wd','filename')` searches for a file within the base working directory.

`sdtweb('_tracker','support',979)` opens a tracker on the support web site.

`sdtweb('_BP','FunctionName','Tag')` Find Tag in FunctionName (result of `sdtweb FunctionName Tag` and set breakup here for debug.

`sdtweb('_TexFromHTML','HmtlFileName')` Find .tex and line source corresponding to the HmtlFileName.html help file.

_taglist

This commands opens the TagList figure (tree view of your file providing links for source code navigation)

```
sdtweb _taglist   % Open taglist of current editor file (if not docked)
sdtweb _taglist feutil % Open taglist of feutil
```

Accepted command options are

- `-sortABC` will display the navigation tree alphabetically sorted.
- `-level`*`val`* in combination with `sortABC` perform the alphabetical sorting up to level *`val`*.

The coding styles convention associated to the `TagList` parsing are detailed in section 7.17 (`sdtweb('syntax')`).

sp_util

Purpose

Sparse matrix utilities.

Description

This function should be used as a `mex` file. The `.m` file version does not support all functionalities, 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 an idea of the memory required to store a Cholesky factor of this matrix.
- `ks=sp_util('sp2sky',sparse(k))` returns the structure array used by the `ofact` object.
- `ks = sp_util('sky_dec',ks)` computes the LDL' factor of a ofact object and replaces the object data by the factor. The `sky_inv` command is used for forward/backward substitution (take a look at the `@ofact\mldivide.m` function). `sky_mul` provides matrix multiplication for unfactored ofact matrices.
- `k = sp_util('nas2sp',K,RowStart,InColumn,opt)` is used by `nasread` for fast transformation between NASTRAN binary format and MATLAB sparse matrix storage.
- `k = sp_util('spind',k,ind)` renumbering and/or block extraction of a matrix. The input and output arguments `k` MUST be the same. This is not typically acceptable behavior for MATLAB functions but the speed-up compared with `k=k(ind,ind)` can be significant.
- `k = sp_util('xkx',x,k)` coordinate change for `x` a 3 by 3 matrix and DOFs of `k` stacked by groups of 3 for which the coordinate change must be applied.
- `ener = sp_util('ener',ki,ke,length(Up.DOF),mind,T)` is used by `upcom` to compute energy distributions in a list of elements. Note that this function does not handle numerical round-off problems in the same way as previous calls.

- `k = sp_util('mind',ki,ke,N,mind)` returns the square sparse matrix `k` associated to the vector of full matrix indices `ki` (column-wise position from `1` to `N^2`) and associated values `ke`. This is used for finite element model assembly by `fe_mk` and `upcom`. In the later case, the optional argument `mind` is used to multiply the blocks of `ke` by appropriate coefficients. `mindsym` has the same objective but assumes that `ki,ke` only store the upper half of a symmetric matrix.

- `sparse = sp_util('sp2st',k)` returns a structure array with fields corresponding to the MATLAB sparse matrix object. This is a debugging tool.

- `sp_util('setinput',mat,vect,start)` places vector `vect` in matrix `mat` starting at C position `start`. Be careful to note that `start` is modified to contain the end position.

stack_get,stack_set,stack_rm

Purpose

Stack handling functions.

Syntax

```
[StackRows,index]=stack_get(model,typ);
[StackRows,index]=stack_get(model,typ,name);
[StackRows,index]=stack_get(model,typ,name,opt);
Up=stack_set(model,typ,name,val)
Up=stack_rm(model,typ,name);
Up=stack_rm(model,typ);
Up=stack_rm(model,'',name);
[model,r1]=stack_rm(model,typ,name,opt);
```

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. If the types are different, this is not an obligation, just good practice.

In get and remove calls, `typ` and `name` can start by `#` to use a regular expression based on matching (use `doc regexp` to access detailed documentation on regular expressions). To avoid selection by `typ` or `name` one can set it to an empty string.

Command options can be given in `opt` to recover stack lines or entries.

- `stack_get` outputs selected sub-stack lines by default.
  - Using `opt` set to `get` or to `GetData` allows directly recovering the content of the stack entry instead of the stack line.
  - Using `opt` set to `multi` asks to return sub stack lines for multiple results, this is seldom used.
- `stack_rm` outputs the model from which stack lines corresponding to typ and name have been removed.
  - Using `opt` set to `get` will output in a second argument the removed lines.

– Using opt set to GetData will output in a second argument the content of the removed lines. If several lines are removed,

Syntax

```
% Sample calls to stack_get and stack_rm
Case.Stack={'DofSet','Point accel',[4.03;55.03];
            'DofLoad','Force',[2.03];
            'SensDof','Sensors',[4 55 30]'+.03};

% Replace first entry
Case=stack_set(Case,'DofSet','Point accel',[4.03;55.03;2.03]);
Case.Stack

% Add new entry
Case=stack_set(Case,'DofSet','P2',[4.03]);
Case.Stack

% Remove entry
Case=stack_rm(Case,'','Sensors');Case.Stack

% Get DofSet entries and access
[Val,ind]=stack_get(Case,'DofSet')
Case.Stack{ind(1),3} % same as Val{1,3}
% Direct access to cell content
[Val,ind]=stack_get(Case,'DofSet','P2','get')

% Regular expression match of entries starting with a P
stack_get(Case,'','#P*')

% Remove Force entry and keep it
[Case,r1]=stack_rm(Case,'','Force','get')
```

SDT provides simplified access to stacks in feplot (see section 4.4.3) and iiplot figures (see section 2.1.2). cf.Stack{'Name'} can be used for direct access to the stack, and cf.CStack{'Name'} for access to FEM model case stacks.

ufread

Purpose

Read from Universal Files.

Syntax

```
      ufread
      ufread('FileNameOrList')
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, `UFF55` analysis data at node, `UFF58` 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`. `ufread('FileName')` opens an `feplot` or `iiplot` figure with the contents. `UFS=ufread('FileName')` returns either a FEM model (if only model information is given) or a curve stack `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 file contents by displaying `UFS`

```
>> UFS

UFS = UFF curve stack 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 stack, 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 an iiplot stack entry using calls of the form ci.Stack{'curve','Test'}=UFS(3).If you need to extract partial sets of DOF, consider fe_def SubDof.

15 Grid point

Grid points stored in a node matrix (see node page 273) 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

UFF55 *Analysis data at nodes* are characterized by poles .po and residues .res (corresponding to DOFs .dof) and correspond to shape at DOF datasets (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 .fun field
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.

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

`UFF58` *Functions at nodal DOF* (see `Response data`) 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`.

`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 — This is stored in .dof field (which also has a file number as address in column 3). Values are

- 2(I5,I10) : FunType (list with xfopt('_funtype'), stored in .fun(1)), FunID (ID in .dof(:,5)), VerID version or sequence number, LoadCase (0 single point)
- (1X,10A1,I10,I4) : ResponseGroup (NONE if unused, ID in .dof(:,4)), ResponseNodeID, ResponseDofID (1:6 correspond to *SDT* DOFs .01 to .06, -1:-6 to *SDT* DOFs .07 to .12). DOF coding stored in .dof(:,1)).
- (1X,10A1,I10,I4) : ReferenceGroup (NONE if unused, ID in .dof(:,4)), ReferenceNodeID, ReferenceDofID. These are only relevant if LoadCase is zero. DOF coding stored in .dof(:,2)).

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

UFF82 *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 curve stack UFS to the file FileName. For details on curve stacks see section 2.1.2 . 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 curve stack DB=xfopt('empty'). You can then copy data sets from the stack 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 curve stack built, ufwrite('NewFile',UF,1:3) will write the three datasets.

With iiplot, you can use the stack to change properties as needed then write selected datasets to a file. For example,

```
tname=nas2up('tempname .uf');
ci=iicom('CurveLoad','gartid');
```

```
ci.Stack{'Test'}.x='frequency'; % modify properties, see xfopt('_datatype')
ci.Stack{'Test'}.yn='accele';
iicom('sub');          % reinitialize plot to check
ufwrite(tname,ci,'Test');
% write a model
ci.Stack{'SE','model'}=demosdt('demo gartte');
ufwrite(tname,ci,'model');
% write a time trace
C1=fe_curve('TestRicker .6 2',linspace(0,1.2,120));
C1=ufwrite('_toxf',C1); % Transform to xf format
C1.x= xfopt('_datatype','time');
C1.yn= xfopt('_datatype','Acceleration');
C1.fun= xfopt('_funtype',1);
ufwrite(tname,ci,'Ricker');
UFS=ufread(tname); % 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 (6.120).

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 5.3.2 .

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 6.4 and section 6.5 .

Commands

Clear, Load *File* , Save *File*

upcom('clear') clears the global variable Up and the local and base variables Up if they exist. If these local variables are not cleared then the global variable Up is reset to that value.

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. You should look up newer assembly calls in section 4.8.8 .

`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 assemble 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` option can be used to prevent the default projection of the matrices on the master DOFs defined by the current case.

The `Point` option 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]`

Given a parameterized model, the command `ComputeModal` computes the frequency response associated to all the inputs and outputs of the model, taken into account the damping ratio. `ComputeModal` computes the normal modes and static corrections for inputs of the full or reduced order models based on the full or reduced model. `nor2xf` is then called to build the responses (for sensor load definitions within the model, see `nor2xf`).

```
Up=upcom('load GartUp');
Up=fe_case(Up,'SensDof','sensors',[3.03;54.03],'DofLoad','input',3.03);
upcom(Up,'compute modal full acc iiplot "updated" -po -reset');
```

You may want to compute the direct frequency response associated the inputs on all the DOFs structure. It does not compute modes and is thus faster than ComputeModal for a full order model and a few frequency points. The high level call uses the fe_simul function

```
cf=fecom('load',which('GartUp.mat'));
cf.mdl=fe_case(cf.mdl,'DofLoad','input',3.03);
cf.mdl=stack_set(cf.mdl,'info','Freq',linspace(0,15,50)');
cf.def=fe_simul('DFRF',cf.mdl);fecom('ch22');
```

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('load GartUp');upcom('plotelt');
cf=feplot;cf.def=fe_eig(Up,[5 10 1e3]);fecom('ch7');
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 ' ')` 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 [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.

```
Up=upcom('load GartUp');  % Load sample model
Up=fe_case(Up,'ParReset') % Reset parameters
Up=fe_case(Up,'ParAdd k 1.0 0.5 2.0','Tail','group3');
Up=fe_case(Up,'ParAdd t 1.0 0.9 1.1','Constrained Layer','group6');
Up=fe_case(Up,'parcoef',[1.2 1.3]);
upcom(Up,'info par');
```

Parameters are stored in the case stack and can be selected with

```
des=fe_case(Up,'stack_get','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 upcom ParCoef).
.pdir	Boolean vector giving the positions of affected elements in Up.mind (for upcom models)
.name	Parameter name
.zCoef	optional string definition of the zcoef associated to the parameter.

ParCoef

The value of each physical parameter declared using upcom Par or fe_case par commands is described by a row of coefficients following the format

```
[type cur min max vtype]
```

with

- type 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 (upcom only). 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 during upcom('SetNominal').

- cur for current value
- min for minimum value
- max for maximum value
- vtype deals with the type of variation 1 linear, 2 log (not fully implemented)

upcom(Up,'parcoef',cur) is used to set current values (cur must be a vector of length the number of declared parameters), while upcom(Up,'parcoef',par) also sets min, max and vtype values. You can also use [cur,par]=upcom(Up,'parcoef') or par=upcom(Up,'parcoefpar') to obtain current values or the parameter value matrix.

An example of parameter setting is thus

```
Up=demosdt('gartup'); % see sdtweb demosdt('gartup')
%      MatType cur min max vtype
par = [ 1      1.0 0.1 3.0  1 ; ... % Linear
        3      0.0  -1 2.0  2 ];    % Log variation
Up=upcom(Up,'parcoef',par);
```

```
upcom(Up,'info par');
[cur,par]=upcom(Up,'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 6.2.7 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.

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 Par 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. [42] 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*]

To generate a new model, you should first clear any Up variable in the workspace, specify the file that where you will want the element matrices to be saved, then perform the assembly. For example

```
model=demosdt('demogartfe');
model.wd=sdtdef('tempdir');model.file='GartUp_demo.mat';
Up=upcom(model,'setnominal')
% delete(fullfile(Up.wd,[Up.file,'.mat'])) % to remove the result
```

Case information (boundary conditions, ... see fe_case) in model is saved in Up.Stack and will be used in assembly unless the NoT option is included in the Assemble command.

If the parameter that will be declared using the Par 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',FEnode,FEelt,pl,il,[],adof) (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.

ki,mi are vectors of indices giving the position of element matrix values stored in ke,me. The indices use the column oriented numbering from 1 to N^2 where N is the assembled matrix size.

Up.mind is a *NElt* x*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 Par 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 Par 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 Par 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 Par command) such that the difference Log least-squares difference (3.4) 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 5.1 .
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 Par 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

Description

Class constructor for variable handle objects.

v_handle

The *Structural Dynamics Toolbox* supports variable handle objects, which act as **pointers** to variables that are actually stored as

- `uo` user data of graphical objects (init with `v_handle('uo',go)`). This is in particular used in `feplot` to store the model in `cf.mdl`. For easier access, the format `v_handle('uo',parent,'tag','TipCh')` allows search by tag and possible creation as a invisible `uicontrol`.

  It is possible to associate a callback executed when the variable is modified using `v_handle('uo',go,SetFcn)`

- `so` reference to another (stored) object.

- `mat` data in files. This latter application may become very useful when handling very large models. `sdthdf` indeed allows RAM unloading by keeping data on drive while using a pointed to it. A trade-off between data access performance (limited to your drive I/O performance) and amount of free memory will occur. Some supported file formats are MATLAB 6 `.mat` files (use `v_handle('mat','varname','filename')`), NASTRAN `.op2,op4` (see `nasread`), ABAQUS `.fil` ...

  For data in files, methods of interest are extraction `def(rows,cols)`, total read `def.GetData` or `def(:,:)`, and matrix multiplication `c*def`.

- `hdf` data in MATLAB ¿7.3 HDF based `.mat` files (see `sdthdf hdfReadRef`)

- `base` global variables (init with `v_handle('global','name')`), use is discontinued

- `mkls` 32 bit sparse (init with `v_handle('mkls',k)`) used for improved time response

`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`.

xfopt

Purpose

User interface for curve stack pointer objects. **Stack, see section 2.1.2 , are now preferred so this function is documented mostly for compatibility.**

Syntax

```
xfopt command
XF(1).FieldName=FieldValue
XF(1).command='value'
XF.check
r1=XF(1).GetData
curve=XF(1).GetAsCurve
XF.save='FileName'
```

Description

SDT considers data sets in curve, `curve` `Response data` or `Shapes at DOFs` formats. Handling of datasets is described in the iiplot tutorial which illustrates the use of curve stacks (previously called database wrappers).

`ufread` and `ufwrite` also use curve stacks which can be stored as variables. In this case, FEM models can also be stored in the stack.

The use of a stack pointer (obtained with `XF=iicom(ci,'curvexf');`) has side advantages that further checks on user input are performed.

`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.

Object saving is overloaded so that data is retrieved from a `iiplot` figure if appropriate before saving the data to a mat file.

Object field setting is also overloaded (consistency checks are performed before actually setting a field) This is illustrated by the following example

```
[ci,XF]=iiplot
XF(1)
XF(1).x='time'; XF(1).x
```

where `XF(1)` is a `Response data` set (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.

_FunType, _DataType, _FieldType

These commands are used internally by SDT. `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.

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] K. McConnell, *Vibration Testing. Theory and Practice.* Wiley Interscience, New-York, 1995.

[3] W. Heylen, S. Lammens, and P. Sas, *Modal Analysis Theory and Testing.* KUL Press, Leuven, Belgium, 1997.

[4] D. Ewins, *Modal Testing: Theory and Practice.* John Wiley and Sons, Inc., New York, NY, 1984.

[5] E. Balmes, *Methods for vibration design and validation.* Course notes ENSAM/Ecole Centrale Paris, 1997-2012.

[6] "Vibration and shock - experimental determination of mechanical mobility," *ISO 7626*, 1986.

[7] 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.

[8] 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.

[9] E. Balmes, "Frequency domain identification of structural dynamics using the pole/residue parametrization," *International Modal Analysis Conference*, pp. 540–546, 1996.

[10] 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.

[11] 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.

[12] E. Balmes, "New results on the identification of normal modes from experimental complex modes," *Mechanical Systems and Signal Processing*, vol. 11, no. 2, pp. 229–243, 1997.

[13] 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.

[14] 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.

[15] E. Balmes, "Review and evaluation of shape expansion methods," *International Modal Analysis Conference*, pp. 555–561, 2000.

[16] E. Balmes, "Sensors, degrees of freedom, and generalized modeshape expansion methods," *International Modal Analysis Conference*, pp. 628–634, 1999.

[17] 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.

[18] E. Balmes, "Optimal ritz vectors for component mode synthesis using the singular value decomposition," *AIAA Journal*, vol. 34, no. 6, pp. 1256–1260, 1996.

[19] D. Kammer, "Test-analysis model development using an exact modal reduction," *International Journal of Analytical and Experimental Modal Analysis*, pp. 174–179, 1987.

[20] J. O'Callahan, P. Avitabile, and R. Riemer, "System equivalent reduction expansion process (serep)," *IMAC VII*, pp. 29–37, 1989.

[21] R. Guyan, "Reduction of mass and stiffness matrices," *AIAA Journal*, vol. 3, p. 380, 1965.

[22] R. Kidder, "Reduction of structural frequency equations," *AIAA Journal*, vol. 11, no. 6, 1973.

[23] M. Paz, "Dynamic condensation," *AIAA Journal*, vol. 22, no. 5, pp. 724–727, 1984.

[24] 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.

[25] E. Balmes and L. Billet, "Using expansion and interface reduction to enhance structural modification methods," *International Modal Analysis Conference*, February 2001.

[26] MSC/NASTRAN, *Quick Reference Guide 70.7*. MacNeal Shwendler Corp., Los Angeles, CA, February,, 1998.

[27] E. Balmes, "Model reduction for systems with frequency dependent damping properties," *International Modal Analysis Conference*, pp. 223–229, 1997.

[28] T. Hasselman, "Modal coupling in lightly damped structures," *AIAA Journal*, vol. 14, no. 11, pp. 1627–1628, 1976.

[29] A. Plouin and E. Balmes, "A test validated model of plates with constrained viscoelastic materials," *International Modal Analysis Conference*, pp. 194–200, 1999.

[30] E. Balmes and S. Germès, "Tools for viscoelastic damping treatment design. application to an automotive floor panel.," *ISMA*, September 2002.

[31] E. Balmes, *Viscoelastic vibration toolbox, User Manual.* SDTools, 2004-2013.

[32] J.-M. Berthelot, *Materiaux composites - Comportement mecanique et analyse des structures.* Masson, 1992.

[33] N. Atalla, M. Hamdi, and R. Panneton, "Enhanced weak integral formulation for the mixed (u,p) poroelastic equations," *The Journal of the Acoustical Society of America*, vol. 109, pp. 3065–3068, 2001.

[34] J. Allard and N. Atalla, *Propagation of sound in porous media: modelling sound absorbing materials.* Wiley, 2009.

[35] A. Girard, "Modal effective mass models in structural dynamics," *International Modal Analysis Conference*, pp. 45–50, 1991.

[36] 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.

[37] 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.

[38] 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.

[39] R. J. Craig and M. Bampton, "Coupling of substructures for dynamic analyses," *AIAA Journal*, vol. 6, no. 7, pp. 1313–1319, 1968.

[40] E. Balmes, "Use of generalized interface degrees of freedom in component mode synthesis," *International Modal Analysis Conference*, pp. 204–210, 1996.

[41] 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.

[42] E. Balmes, "Efficient sensitivity analysis based on finite element model reduction," *International Modal Analysis Conference*, pp. 1168–1174, 1998.

[43] T. Hughes, *The Finite Element Method, Linear Static and Dynamic Finite Element Analysis*. Prentice-Hall International, 1987.

[44] H. J.-P. Morand and R. Ohayon, *Fluid Structure Interaction*. J. Wiley & Sons 1995, Masson, 1992.

[45] J. Imbert, *Analyse des Structures par Eléments Finis*. E.N.S.A.E. Cépaques Editions.

[46] 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.

[47] 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.

[48] 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.

[49] E. Balmes, "Orthogonal maximum sequence sensor placements algorithms for modal tests, expansion and visibility.," *IMAC*, January 2005.

[50] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[51] 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.

[52] 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.

[53] G. Vermot Des Roches, *Frequency and time simulation of squeal instabilities. Application to the design of industrial automotive brakes*. PhD thesis, Ecole Centrale Paris, CIFRE SDTools, 2010.

[54] M. Jean, "The non-smooth contact dynamics method," *Computer methods in Applied Mechanics and Engineering*, no. 177, pp. 235–257, 1999.

[55] 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.

[56] N. Lieven and D. Ewins, "Spatial correlation of modeshapes, the coordinate modal assurance criterion (comac)," *International Modal Analysis Conference*, 1988.

[57] D. Hunt, "Application of an enhanced coordinate modal assurance criterion," *International Modal Analysis Conference*, pp. 66–71, 1992.

[58] R. Williams, J. Crowley, and H. Vold, "The multivariate mode indicator function in modal analysis," *International Modal Analysis Conference*, pp. 66–70, 1985.

[59] 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.

[60] 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.

Index

.ID, 794

, 497

actuator dynamics, 748
addresses, 290
adof, 292
AMIF, 789
animation, 156, 506
AnimMovie, 161
assembly, 671
asymptotic correction, 738
attachment mode, 245, 701
automated meshing, 146

b, 200, 609, 663
bar element, 386
beam element, 387
boundary condition, 166, 610
BuildConstit, 325

c, 200, 609
Case.GroupInfo, 308
cases, 284, 612
cf, 150, 589, 770, 870
channel, 288
CMIF, 789
collocated, 105, 212, 751
color mode, 508
ColorMap, 510, 797
COMAC, 122, 781
command formatting, 489
command function, 489
complex mode
    computation and normalization, 631
    definition, 210
    identification, 96, 747, 751
Complex Mode Indicator Function, 789
Component Mode Synthesis, 247, 633
connectivity line matrix, 97, 873
coordinate, 273, 475
cost function
    logLS, 774
    quadratic, 96, 774
cp, 202
Craig Bampton reduction, 246, 698
Cross generalized mass, 782
curve, 286
curve stack, 46, 890
Cyclic symmetry, 643

damping, 203
    non-proportional, 108, 205, 742
    proportional or modal, 107, 204, 840
    Rayleigh, 205
    structural, 205, 206, 816
    viscoelastic, 205, 206
damping ratio, 284
data structure
    case, 284
    curve, 286
    deformation, 285
    element constants, 309
    element property, 277
    GroupInfo, 308

material, 276
model, 280
sens, 623
database wrapper, 46, 50, 64, 870, 890
def, 285
DefaultZeta, 853
degree of freedom (DOF), 201
active, 609, 631, 650
definition vector, 278, 290, 292, 609
element, 279
master, 301
nodal, 278
selection, 292, 609
demonstrations, 12
design parameters, 258
DID, 273, 392, 476
dirp, 262
dock, 483
drawing axes, 762

effective mass, 244
EGID, 275, 279, 297
eigenvalue, 189, 631, 650
element
bar, 386
beam, 387
EGID, 275, 279
EltID, 325
fluid, 396, 398
function, 274, 311, 321, 594
group, 274, 518
identification number (EltId), 279
plate, 394, 444, 447, 451
property row, 275, 409, 438, 668
rigid link, 390, 448
selection, 296, 518, 528, 546
solid, 399
user defined, 311
EltId, 275
EltOrient, 299
eta, 206, 284
expansion, 132, 654

family of models, 258
FE model update, 265–267
based on FRFs, 888
based on modal frequencies, 887
command function, 878
FEelt, 144, 524
FEMLINK, 460, 471, 498
FEMLink, 805, 831, 846
FEnode, 144, 524
feplot, 139, 149, 589
frequency
unit, 650
frequency response function (FRF), 214, 747
frequency units, 738, 748, 839, 885
frequency vector w, 215, 748

ga, 202
generalized mass, 189, 243, 735
GID, 273
global variable, 17, 144, 524, 538, 594
Guyan condensation, 246, 698

hexahedron, 406

identification, 54
direct system parameter, 76, 740
minimal model, 103, 105, 751
normal mode model, 742
options, 737
orthogonal polynomials, 77, 746
poles, complex mode residues, 96, 747
poles, normal mode residues, 107, 747
reciprocal model, 751
scaled modeshapes, 108, 751
IDopt, 66, 291, 737, 890
iiplot, 44, 770
IIxf, 49, 50, 74
il, 277

importing data, 63, 65, 146
ImWrite, 161
input shape matrix b, 200, 609
integinfo, 325
isostatic constraint, 701

LabFcn, 513, 686
load, 200, 663
localization matrix, 201
loss factor, 206, 284

MAC, 122, 775, 777
MACCO, 122, 780
Map, 548
mass
  effective, 244
  generalized, 243
  normalization, 108, 243, 650, 677
material function, 276
material properties, 276, 668
MatID, 509
MatId, 275, 276, 297, 325
matrix
  ofact, 826, 866
  sparse/full, 826, 866
mdof, 278
meshing, 146
MIMO, 103
minimal model, 103, 751
MMIF, 788
modal
  damping, 107
  input matrix, 204, 210
  mass, 189, 243, 735
  output matrix, 204, 210
  participation factor, 212
  scale factor, 783
  stiffness, 243
Modal Scale Factor, 782
mode
  acceleration method, 245
  attachment, 701
  complex, 210, 631
  constraint, 698
  displacement method, 245
  expansion, 132, 654
  normal, 242, 650
  scaling, 211, 243
model, 280
  description matrix, 274
  reduction, 501
multiplicity, 103, 752
Multivariate Mode Indicator Function, 788

NASTRAN, 805, 810
node, 139, 273
  group, 273
  selection, 273, 293, 528, 546
NodeId, 273
nor, 202, 816
normal, 548
normal mode
  computation and normalization, 650
  definition, 242
  format, 202
  identification, 108, 742
  model, 816
  residue, 107
NoT, 301
notations, 17

object
  ofact, 826
  sdth, 855
  `v_handle`, 889
observation, 200
om, 202
orientation
  triax, 523
orthogonality conditions, 243, 631, 650, 677

output shape matrix c, 200, 609

ParamEdit, 331
pb, 202
pentahedron, 405
PID, 273, 475
pl, 276
plate element, 394, 444, 447, 451
po, 801
POC, 122, 782
pole, 212, 244
    formats, 801
    lines, 770, 794
    multiplicity, 103, 751
pole residue format, 212
polynomial model format, 214
ProID, 509
ProId, 275, 277, 297, 325
property function, 277

quadrilateral, 403

Rayleigh, 205
reciprocity, 105, 211, 610, 751
reduction basis, 241, 698
renderer, 516
res, 212, 840, 841
residual
    dynamic, 128–130
    high frequency, 212, 244
    low frequency, 212
residue matrix, 103, 107, 204, 210, 213
rigid body modes, 245, 698
rigid link, 390, 448

scalar spring, 390
scaling, 517, 751, 779
scatter, 772
segment, 401
selection
    element, 296
    node, 293
sensor, 168
    dynamics, 748
    placement, 120, 703
simulate, 187
solid element, 399
sparse eigensolution, 650
ss, 209, 838
stack, 16, 281
stack entries, 281
state-space models, 209, 838, 841
static correction, 192, 212, 244, 245, 501
static flexible response, 701
structural modification, 134
subplot, 522, 762
superelement
    command function, 594

tempdir, 853
test/analysis correlation, 703
tetrahedron, 404
tf, 214, 838, 844
time-delays, 748
triangle, 402
two-bay truss, 139

UFS, 870, 876
Universal File Format, 870

VectMap, 310
vector correlation, 775
view, 767, 769

wire-frame plots, 97, 525, 541, 873

XF, 50, 871, 890
xf, 214, 890
XFdof, 51

zeta, 204, 284