# SDT HBM non-linear toolbox

User guide

Partially funded by CLIMA a project supported by Conseil régional d'Ile-de-France

Etienne Balmes Guillaume Vermot des Roches 
 SDTools

 44 rue Vergniaud

 75013 Paris (France)

 Tel.
 33 +1 44 24 63 71

© Copyright 1991-2019 by SDTools.

# Contents

1	The	eory and reference						
	1.1	Kinematics of non-linear systems						
		1.1.1 Observation of strains, stresses, application of forces						
		1.1.2 Manual definition of input and output						
		1.1.3 Generalized non-linear springs						
		1.1.4 Generalized non-linear surfaces						
		1.1.5 Non-linear volumes						
	1.2	Non-linear constitutive laws						
		1.2.1 Laws with no internal states, principles						
		1.2.2 Tabular interpolation						
		1.2.3 Dedicated user function						
		1.2.4 .anonymous field for definition 1						
		1.2.5 Laws with internal states 1						
		1.2.6 Maxwell cells model as an example						
		1.2.7 Iwan						
1.3 Transient solution of non-linear equations								
		1.3.1 Principles						
		1.3.2 Definition of an underlying linear system						
	1.4	Data structures for non-linearities in time						
		1.4.1 NLdata non-linearity definition 2						
		1.4.2 Additional fields in model						
		1.4.3 NL structure : non-linearity representation during time integration 2						
	1.5	Harmonic balance solutions and solver						
		1.5.1 Elastic representation as superelements						
		1.5.2 Time/frequency representation of solutions/loads						
		1.5.3 Harmonic balance equation (real DOF)						
		1.5.4 Complex formulation and equivalent stiffness						
		1.5.5 Inter-harmonic coupling discussion						
		1.5.6 Load definition $\ldots \ldots 3$						

# CONTENTS

	1.6	Data s	structures for HBM solvers
		1.6.1	Model, superelement
		1.6.2	Non-linearity definition NLdata
		1.6.3	NL structure non-linearity representation during HBM solve
		1.6.4	Solver options definition
		1.6.5	Option structure during HBM solve
		1.6.6	Harmonic result structure
2	Tut	orial	45
_	2.1	Install	ation
	2.2	Freque	ency domain test cases
		2.2.1	Example lists
		2.2.2	Spring mass examples
		2.2.3	Beam problem with local non-linearity
		2.2.4	Lap joint problem with contact
		2.2.5	Hyperelastic bushing
	2.3	Time	domain test cases
	-	2.3.1	Single mass test of various non-linearities
		2.3.2	Single mass stepped sine
		2.3.3	CBush with orientation
		2.3.4	Beam with non-linear rotation spring
		2.3.5	Lap joint with non-linear springs
	2.4	NAST	RAN interfacing $\ldots \ldots \ldots$
		2.4.1	Non-linear springs
		2.4.2	Superelement generation (Craig-Bampton type)
		2.4.3	Superelement generation (Free mode variants)
		2.4.4	Storing SDT-HBM options in bulk format
		2.4.5	Data export
	2.5	ABAG	DUS interfacing
		2.5.1	Non-linear springs
		2.5.2	Superelement generation (Craig-Bampton type)
	2.6	ANSY	S interfacing $\ldots$
		2.6.1	Non-linear springs
		2.6.2	Superelement generation (Craig-Bampton type)
	2.7	Avanc	$ed$ usage $\ldots$ $\ldots$ $\ldots$ $\ldots$ $\ldots$ $59$
		2.7.1	DOE & general organisation in steps
		2.7.2	Obsolete NLVibKit
3	Fun	ction 1	ceference 63
			d_hbm 65

# CONTENTS

hbmui										66
hbm_post										73
hbm_solve										77
hbm_utils										85
nl_spring										87
mkl_utils										99
chandle										102
Non linearities list		_								104
Creating a new non linearity: nl_fun.m										119
nl_solve										122
nl_mesh										130
spfmex_utils										137

# Bibliography

3

139

# CONTENTS

## Contents

1.1	Kine	ematics of non-linear systems	7
	1.1.1	Observation of strains, stresses, application of forces	7
	1.1.2	Manual definition of input and output	9
	1.1.3	Generalized non-linear springs	9
	1.1.4	Generalized non-linear surfaces	11
	1.1.5	Non-linear volumes	11
1.2	Non	-linear constitutive laws	<b>13</b>
	1.2.1	Laws with no internal states, principles	13
	1.2.2	Tabular interpolation	13
	1.2.3	Dedicated user function	14
	1.2.4	anonymous field for definition	15
	1.2.5	Laws with internal states	16
	1.2.6	Maxwell cells model as an example	19
	1.2.7	Iwan	20
1.3	Trar	sient solution of non-linear equations	<b>21</b>
	1.3.1	Principles	21
	1.3.2	Definition of an underlying linear system	23
1.4	Data	a structures for non-linearities in time	<b>24</b>
	1.4.1	NLdata non-linearity definition	24
	1.4.2	Additional fields in model	25
	1.4.3	NL structure : non-linearity representation during time integration	26
1.5	Har	monic balance solutions and solver	<b>27</b>
	1.5.1	Elastic representation as superelements	28
	1.5.2	Time/frequency representation of solutions/loads	28
	1.5.3	Harmonic balance equation (real DOF)	30
	1.5.4	Complex formulation and equivalent stiffness	32
	1.5.5	Inter-harmonic coupling discussion	34

1

1.5.6	Load definition	35
1.6 Dat	a structures for HBM solvers	<b>37</b>
1.6.1	Model, superelement $\ldots$	37
1.6.2	Non-linearity definition NLdata	39
1.6.3	NL structure non-linearity representation during HBM solve	39
1.6.4	Solver options definition	40
1.6.5	Option structure during HBM solve	42
1.6.6	Harmonic result structure	43

## **1.1** Kinematics of non-linear systems

#### 1.1.1 Observation of strains, stresses, application of forces

One is interested in solving equations of the general form

$$[M] \{ \ddot{q} \} + [C] \{ \dot{q} \} + [K] \{ q(t) \} = F_{NL}(q, \dot{q}, q_{NL}, \omega, t) + F_{ext}(\omega, t)$$
(1.1)

with q the finite element DOFs, M,C,K the mass, viscous damping, and stiffness matrices respectively,  $F_{ext}$  the external forces and  $F_{NL}$  the non-linear forces internal to the system which may depend on the FEM motion described by its DOFs q and their velocities  $\dot{q}$  as well as possibly internal states of the non-linear elements  $q_{NL}$ . When internal states are necessary (friction, plasticity, ...), additional equations are provided to model their evolution.

Estimation of the non-linear forces can, in a very general fashion, be decomposed in three steps: observation of strains, evaluation of constitutive law at a material point to compute stresses, application of stresses on the model as detailed below.

• Observation of non-linear strains is the first step

$$\{\epsilon(t)\} = [c] \{q(t)\}$$
(1.2)

where strains may represent quantities dependent on the model displacement (relative motion, mechanical strain, ...), but also possibly internal states of the non-linearity if those are included in the definition of q. For HBM solutions  $q_{NL}$  will be assumed to be part of q but for transient simulations this may not be optimal.

Strategies for the construction of the observation matrix c will be discussed for point to point connection in section 1.1.3, surfaces in section 1.1.4, volumes in section 1.1.5.

In the time domain, generalized strains  $\epsilon$  (noted .unl in the code) are obtained by computing

$$u_{nl} = [c] \{q\} + u_{nl0} \tag{1.3}$$

In the implementation, the strain vector may have  $N_E$  components  $e_i$ , and strains at  $N_G$  material points (Gauss or physical points) may be stored as a single vector to allow vectorization of non-linearities of a given kind, thus leading to  $e_{i,gk}$  components.

For HBM computations  $N_T$  times will be evaluated and stored as columns. The internal storage

in field NL.unl is thus of the form

$$\{u_{nl}\}_{(N_E \times N_G) \times N_T} = \{\epsilon\}_{\text{gauss repeat} \times \text{time}} = \begin{bmatrix} e_{1,g1}(t_1) & e_{1,g1}(t_2) & \dots \\ e_{2,g1}(t_1) & \dots & \\ \vdots & & \\ e_{1,g2}(t_1) & \dots & \\ \vdots & & \\ \vdots & & \\ \end{bmatrix}$$
(1.4)

In a similar fashion strain rates are obtained using

$$v_{nl} = [c] \{q'\} + v_{nl0} \tag{1.5}$$

and stored in field NL.vnl.

• From the observed strains, a constitutive law is used to estimate stresses (or generalized forces), as will be discussed in section 1.2 from data present in the NL.Fu field,

$$\{s_{nl}(t)\} = F\left(\{\epsilon(t)\}, \{\dot{\epsilon}(t)\}\right) \tag{1.6}$$

The generic representation of non-linearities should verify classical assumptions on objectivity and on the validity of constitutive relations, which is compatible with the idea that generalized strains are defined at a material point.

The definition of a non-linear constitutive relation giving a definition of generalized stresses  $s_{nl}$  which has the same  $(N_E \times N_C) \times N_T$  size as  $u_{nl}$  and is thus stored NL.unl field to allow overwrite. This allows memory optimization even though the output result is a force and not a strain). When internal states are present, the strain field NL.unl may not be efficiently used to store stresses, it is then possible to store stresses in field NL.snl of size  $(N_S \times N_G) \times N_T$  with  $N_S$  the number of stress components.

• Finally stresses can be applied on the model to **obtain the model forces** in the discretized model associated with DOFs q. That is

$$\{-F_{nl}\} = [b] \{s_{nl}\}$$
(1.7)

where the field NL.snl may be defined or, in cases with no internal states, it is possible to optimize memory by storing  $s_{nl}$  in NL.unl.

In the proposed framework,

#### 1.1. KINEMATICS OF NON-LINEAR SYSTEMS

- internal states of a non-linearity are expected be included within the generalized strains. For a time computation unl will thus be of size  $(N_E + N_{int}) \times N_g$  ( $\times N_T$  in the case of HBM computations). See section 1.2.5 for more details.
- It is generally useful to store current  $u_{nl}(t_n)$ , initial  $u_{nl0}$  and previous  $u_{nl}(t_{n-1})$  values as consecutive blocks in NL.unl(:,:,1),NL.unl(:,:,2),NL.unl(:,:,3).
- In many instances the observation c and command b matrices can be considered constant, but for contacts with large relative displacements they may need to be considered as operators depending on the model state and thus modified dynamically using C code.

When performing a residual call, it is optimal from a performance point of view to perform updating of steps. xxx

## 1.1.2 Manual definition of input and output

While the general approach is to associate non-linearities with strains in elements as described in the following sections, it is possible to define a pro.NLdata structure giving

- .b,.c: manual command and observation definition. If the NLdata.DOF field is defined, placement of the observation matrix in done during the model assembly phase assuming .DOF correspond to mdof (projection with the Case.T matrix, c \* T, is done). If one want to define NL on active DOF, one will provide .adof field (rather than .DOF field): then c is assumed to be defined on active dof, and a simple placeindof is done. If there is no NLdata.DOF field, .c and .b matrices are assumed to be on mdof, and projected using Case.T. For storage during solves, see .b.
- .Sens,.Load Alternate form for .b, .c giving command and observation matrices as .Sens cell array of the form {SensType,SensData} where SensType is a string defining the sensor type and SensData a matrix with the sensor data (see sdtweb sensor). .Load data structure defining the command as a load (with .DOF and .def fields).

## 1.1.3 Generalized non-linear springs

The simplest example is the observation of unaxial springs oriented along vector  $\{d\}$  where the generalized strain is the relative displacement

$$dq = \begin{bmatrix} -\{d\}^T \ \{d\}^T \end{bmatrix} \begin{cases} u_1 \\ v_1 \\ w_1 \\ u_2 \\ v_2 \\ w_2 \end{cases}$$
(1.8)

The non-linear implementation of the OpenFEM cbush element provides 6 generalized strains at a given location corresponding to relative translations and rotations with respect to a local basis  $B = [x_e \ y_e \ z_e]$ 



Initialisation of non-linear behavior in a cbush element group is performed when the NLdata property contains fields

- type='nl\_inout' to let hbm\_solve InitHBM build the needed observation matrix
- Fu='@UserFun' references a user function computing the non-linear force with the prototype call described in section 1.2.3

• .isens can be used to apply non-linearity in some directions only.

A sample problem with cbush can be found in section 2.3.3.

### 1.1.4 Generalized non-linear surfaces

Zero thickness elements which provide 3 generalized strains which correspond at each gauss point to the relative motion of two surfaces in the normal and two tangential directions a xxx Need document

Contact elements have an objective similar to that of zero thickness element but allow the handling of non-conform meshes.

xxx Need document

### 1.1.5 Non-linear volumes

SDT implements observation of strains in configurations where no geometry updating is needed using **StressCut** calls. This strategy is compatible with all physics (acoustics, piezo-electricity, ...) and can also be used for shells (to observe membrane strains and curvature) and beams (to observe axial elongation, shear, torsion and curvature).

.ctype='stresscut' builds the observation matrix for xxx

The generic implementation of multi-physic non-linear volumes provides a generalized strain giving the displacement and its gradient for each field. For a field with 3 components at N nodes

$$\{e_{MP}\}_{12} = \begin{cases} u_1 \\ u_{1,j} \\ u_2 \\ u_{2,j} \\ \vdots \\ \end{bmatrix}_{12} = \begin{bmatrix} N & 0 & 0 \\ N, x & 0 & 0 \\ N, y & 0 & 0 \\ 0 & N, x & 0 \\ 0 & N, y & 0 \\ 0 & N, z & 0 \\ 0 & 0 & N, z & 0 \\ 0 & 0 & N, z & 0 \\ 0 & 0 & N, z \\ 0 & 0 & N, x \\ 0 & 0 & N, y \\ 0 & 0 & N, z \\ \end{bmatrix} \begin{cases} u_1(n1) \\ u_2(n1) \\ u_3(n1) \\ \vdots \\ \end{bmatrix}_{(3N)}$$
(1.10) 
$$= [B_{MP}(r, s, t)]_{12 \times 3N} \{u_{in}\}_{(3N)}$$

As a an example, the standard linearized mechanical strain

$$\begin{cases} \epsilon_{x} \\ \epsilon_{y} \\ \epsilon_{z} \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \end{cases} = \begin{bmatrix} N, x & 0 & 0 \\ 0 & N, y & 0 \\ 0 & 0 & N, z \\ 0 & N, z & N, y \\ N, z & 0 & N, x \\ N, y & N, x & 0 \end{bmatrix} \begin{cases} u \\ v \\ w \end{cases} = [B_{Ep}]_{6 \times 3N} \{u_{in}\}_{3N}$$
(1.11)

can be related to the generic multi-physic strain using a transformation matrix  $T_{Ep}$ 

The usual mechanical stiffness can thus be reformulated using  $B_{MP}$ 

$$K_e = \int_{\Omega} \left[ B_{Ep} \right]^T \left[ D \right] \left[ B_{Ep} \right] d\Omega = \int_{\Omega} \left[ B_{MP} \right]^T \left[ T_{Ep}^T D T_{Ep} \right] \left[ B_{MP} \right] d\Omega$$
(1.13)

Similarly for right hand side evaluations

XXX

A generic non-linear multiphysic constitutive law is a function that will receive a NL structure with fields

- .unl a series of generic strain  $e_{mp}$  at each gauss points thus a matrix with 4\*Nf\*Ng rows (4 components per field, repeated for each gauss point), and possibly multiple columns for different times.
- .vnl a series of strain velocities (time derivative of strain)
- .nodeG matrix of size Nfield \* Ng containing interpolated fields (typically positions x, y, z, un1 but possibly also v1x,v1y,v1z for first orientation axis), .nodeEt is an int32 vector coding the name of each field.
- .MatType list of matrix types for which the tangent consitutive law is given.
- .constit contains the constant parameters read from model.pl.

From this information the function should return a structure with field

- .unl generic stresses a matrix with 4\*Nf\*Ng rows and Nt columns. The name should really be .fnl but the same field name .unl is used to allow memory reuse.
- .MatType propagated version of input field.
- .DD a matrix of size ones(4\*Nf,4\*Nf,Ng,length(NL.MatType)) giving the tangent constitutive law for each Gauss point and each desired matrix type. This corresponds to the  $T_{Ep}^T DT_{Ep}$ in (??).

# 1.2 Non-linear constitutive laws

While implementation of non-linear kinematics is fairly generic, users typically want to implement their own constitutive laws relating stresses to strains and possibly their history.

# 1.2.1 Laws with no internal states, principles

Constitutive laws where the stress only depends on strain and strain rate are the simplest. These laws exploit the framework provided by nl\_inout for various element types. The only thing that needs to be implemented is a .Fu function

$$\{s_{nl}(t)\} = F(u_{nl}, v_{nl}) \tag{1.14}$$

During time/frequency evaluations it is essential that such evaluations be very fast, this has an impact on implementation and different strategies are implemented

- tabular section 1.2.2 .
- dedicated user function section 1.2.3.
- $\bullet\,$  anonymous function defined with parameters section 1.2.4 .

# 1.2.2 Tabular interpolation

```
sdtweb('_eval','d_fetime.m#BumpStop')
opt=d_fetime('timeopt dt=1e-4 tend=.1');opt.Method='Back';
model=fe_time(opt,mdl)
```

Multiple forms are supported. Currently a cell array of

• tabulated Fu, ie 1st column is the observed strain (relative displacement), 2nd column is the corresponding stress (load). A third column is interpreted as a coefficient applied to computed

non linear load associated to velocity (Fv) (it is used for example in bumpstops to take in account a damping with offset on the displacement).

When storing constitutive laws in tabular form, it is desirable to follow the SDT curve format giving the the variable names on which the function depends in the .Xlab fields, abcissa in the .X and values in the .Y field.

- numeric curve ID for a curve in the stack.
- string defining a predefined law, listed using nl\_spring('guilist').

Supported laws are

- BumpStop dp dp kp kp cp cp dm dm km km cm cm (a bumpstop example can be found in sdtwebt\_nlspring('BumpStop'). dp is the upper gap of the bumpstop, kp the upper stiffness, and cp the upper damping, then dm km and cm the same for the lower gap. This bump stop law (as friction law) is built in nl\_spring Tab which is the historical implementation : there is a known approximation, the stiffness is applied from du = dp \* 1.001 to du = 1 (so that the slope is not exactly kp), and damping from du=dp. The same for lower gap. (XXXJP : we should rewrite the nl\_spring Tab command directly in nl\_inout, without this approximation. This approximation was done because of interpolation in third column, that was wanted to be 0 or 1, so that we cannot begin the damping and the stiffness from the same value... we should use a lower du interval than 0.001 \* dp... xxxjp).
- Friction f co co, where f is the friction load, and co is a damping coefficient applied on the transition around 0 velocity (co is typically important). Dry frictions are known to be responsible for convergence problems.
- gapCyll inner cylinder within an external sleeve.
- ctcFric xxx penalized contact and friction implementation

#### 1.2.3 Dedicated user function

The easiest conceptual way to define a non-linearity is to use your own function. For example, if you have defined the function

You can specify that this function should be used to compute a law with no internal states using

```
model=d_fetime('TestbeamNL');
model=nl_spring('SetPro proid100 Fu="@resCubic"',model);
% verify that Fu was defined in NLdata
NL=stack_get(model,'','NL','get');NL.NLdata.Fu
% The same with a subfunction in d_hbm
model=nl_spring('SetPro proid100 Fu="d_hbm(''@resCubic'')"',model);
NL=stack_get(model,'','NL','get');NL.NLdata.Fu
```

Note that in instances of deployed MATLAB generated with the MATLAB compiler, all custom functions must be defined a priori. And only anonymous functions may be created. The NLdata entry is kept as the one defined by the nl\_inout db call. Entry NLdata.Fu must then be replaced by the handle to the dedicated function in the non-linear property

# 1.2.4 .anonymous field for definition

To allow parameter edition, the base mechanism to automatically create an .Fu as MATLAB anonymous function handle is to use the following NLdata fields

• .anonymous a string which upon model initialization in nl\_spring('Init') will be transformed to an anonymous function of the form

Fu=@(NL,~,~,~,~,~,opt,~)AnonymousString

The  $Q(\ldots)$  part can be included in the string if it differs from the default.

Parameters are fields of the NL non-linear structure (in the example below NL.par1 will be defined). The initialization of these parameters is performed using the NLdata.Param field described below.

In frequency domain solvers, the current frequency (rad/s) is accessible in opt.w.

- .csv a string to define parameters in the anonymous function. This strign declares parameters using cingui ParamEdit format. This declares the parameters to be used, with a default value, their type and a possible brief explanation, as illustrated below.
- .Param The current parameters. .Param can be
  - a string defining the parameters declared in the .csv by par1=val1 par2=val2 ...
  - a structure with fields corresponding exactly to the declared parameters struct('par1', val1, 'par2', val2).

Any omitted parameter will be set to its default declared in .csv. Lack of default values would then results in an error at the function execution.

• .tex a string providing a tex format of the formula used in .anonymous.

Use of inline functions. One can directly use the existing framework with a customized call based on the concept of *anonymous function handles* in MATLAB. The inline function used in NLdata.anonymous has access to xxx

```
d_hbm('TestDuffing2Dof-an')
% completes the definiton
NLdata=struct('csv','par1(1#%g#"value of parameter 1")',...
'Param','par1=val',...
'tex','p_1 u_{NL}'
'anonymous','-NL.par1*NL.unl');
model=feutil('setpro 2001',model,'NLdata',NLdata);
```

#### 1.2.5 Laws with internal states

Classical rheologic model exploit internal states to account for hysteresis phenomena. E.g. The elasto-visco-plastic behavior is shown in figure 1.1 In this case one internal state is required to



Figure 1.1: Elastic-visco-plastic model

represent the relative force between both extremities, to keep track of the relative displacement between the middle spring and middle dashpot.

Very complex models can be associated to this kind of representation, where one will implement internal dynamics associated to a strain history. The general formulation of such system is given as

$$f_{NL} = \mathcal{F}(\{q\},\{\dot{q}\},\{z\},\{\dot{z}\},t)$$
(1.15)

and the internal states evolution equation functional

$$\{\dot{z}\} = \mathcal{G}(\{q\},\{\dot{q}\},\{z\},t)$$
(1.16)

where  $\{z\}$  is a vector of internal states,  $\{q\}$  the displacement vector and  $\dot{x}$  represent the time derivatives.

Equations (1.15) and (1.2.5) represent the class of so-called state space models (for example in the 80's for geophysics applications, in particular by Rice and Ruina [?]).

Internal state representation can be based on the need for an efficient implementation and on the fact that the first order dynamics laws defined by equations (1.15) and (1.2.5) do not comply with a second order based resolution framework (the internal states acceleration is seldom defined).

Equation (1.2.5) can thus be resolved separately, possibly with a sub-integration scheme and an adequate interpolation of the external states.

In the non-linearity framework internal states are stored in the continuity of the generalized strains per Gauss points in field .unl. A single non-linearity only handles a single topology and a single internal model, so that each Gauss point has the same number of strain observations and internal states. The field .unl is then of size  $((N_E + N_I) \times N_G) \times N_T$ . The internal storage in field NL.unl is thus of the form

$$\{u_{nl}\}_{((N_E+N_I)\times N_G)\times N_T} = \{\epsilon\}_{\text{gauss repeat}\times \text{time}} = \begin{bmatrix} e_{1,g1}(t_1) & e_{1,g1}(t_2) & \dots \\ e_{2,g1}(t_1) & \dots & \\ \vdots & & \\ e_{1,g2}(t_1) & \dots & \\ \vdots & & \\ e_{1,i1}(t_1) & & \\ \vdots & & \\ \end{bmatrix}$$
(1.17)

This formalism keeps vectorization capability per instant. The internal rate states are stored in the same manner.

Strain history is stored in the third dimension .unl(:,:,jh). In general for time integration, one will use

- .unl(:,:,1) to store the current strains of the form
- .unl(:,:,2) for the initial strains  $u_{nl0}$
- .unl(:,:,3) to store  $u_{nl,t-1}$  the strains at the previous step

When performing a **residual** call, it is efficient to combine time stepping and state update. The **StoreType** parameters controls what the **StoreState** C function does.

- StoreType=01 single step computation of residual and update of internal states in  $u_{nl}$ . The memory buffer of .unl(:,:,1) is first filled using but assuming that internal states have not changed. The residual function computes a step replacing the data in .unl(:,:,1). If a xxx
- StoreType=02 not yet used.
- StoreType=1.. if the hundreds digits is set to 1, .vnl is assumed not used and thus the associated buffer is emptied.

When .iopt(1) is positive, states assumed to be consecutively stored in model.FNL assuming the field exists. When .iopt(2) is positive, internal states are copied consecutively to the displacement vector u. This is done for Ng=.iopt(5) gauss points while skipping Nstrain=.iopt(3) components and copying Nistate=.iopt(4).

only the previous state is needed to integrate internal state evolutions, as offset  $u_{nl0}$  is already stored in the third dimension,  $u_{nl,t-1}$  is found in the third index.

DOF representation of internal states

The number of internal states depends on the model and thus must be declared in the non-linearity. One must then declare in NLdata the fields .adofi and field .MatTyp to obtain a proper initialization of .unl field and associated observations in nl\_inout.

• Internal states are defined independently for each observation line used in the non-linearity. e.g. for a cbush six directions are available relative to the 3 translations and 3 rotations that can be observed. .adofi is then a line cell array of length the number of observations (6 here). Each cell defines a number of internal states associated to the corresponding observation index by providing a column vector with as many lines as internal states used each containing the DOF extension .99. The cell is left empty if no internal state is declared for a particular direction.

Thus NLdata.adofi={[];[];[];[];.99} will add an internal state to the  $6^{th}$  observation of the non-linearity.

- Field .adofi must be either a one column vector/cell (Gauss point wise replication is supported) or should feature as many columns as expected Gauss points for the non-linearity.
- For a volume element in large tranformation, 9 components of stress gradient are observed. .adofi=zeros(18,1)+.99 will interlace 18 internal states with the observation at each gauss point. xxx store internal states as DOF or within NL xxx

By default, one should let intialization procedures allocate DOF identifiers to each internal states by only providing a DOF extention in .adofi. If internal states have a physically defined nodal support, it is also possible to provide the corresponding DOF instead of just an extention. Beware that these DOF should not be coupled to the elastic model, as external resolution would interfere with the internal dynamics. Internal DOF replication for each Gauss point is automatically carried out if a single column is provided.

To allow clean representation and access to internal states, the global model DOF are automatically augmented with DOF associated to internal states. One can then decide whether to keep them or not during the resolution phase by setting positive (kept) or negative (eliminated) signs to the non-empty values in .adofi. To simplify for a given non-linearity all internal states are either kept or not, any negative value will then switch to elimination. In the case where internal states are kept during the resolution displacement and velocity states are automatically updated in the model. HBM solvers specificity

HBM formulations have a resolution approach that is different from usual transient simulations that usually require to write separate dedicated functions for both resolution strategies.

In transient simulations, strain history is available, so that one first integrates internal states defined by equation (1.2.5), and then computes the non-linear forces with equation (1.15).

In HBM based formulations the steady state response state is assumed in the prediction/correction scheme, including strain history. Internal states coefficients are thus predicted, so that the corresponding non-linear force defined inequation (1.15) is directly obtained. One then updates the internal states evolution equation (1.2.5) for convergence iterations. Internal states DOF must thus be kept in the resolution phase.

Two internal states are needed to represent the relative xxx Laws with internal states xxx xxx LS Local slider oscar fun.tex

## 1.2.6 Maxwell cells model as an example

The Maxwell cells model belongs to a category of so-called *rheology* based models. the force at each Gauss point is calculated based on an internal rheology identified as a spring-mass based model. Figure 1.2 illustratesxxx



Figure 1.2: Standard viscoelastic model (single cell Maxwell or Zener). Generalized multi-cell model.

Since each branch is decoupled, one can write a series of scalar internal state evolution equations

$$\dot{e}_n = \frac{K_n}{C_n} (e_n - e_b) \tag{1.18}$$

and recompose the total non-linear stress using

$$s_{nlb} = K_0 e_b + \sum_n K_n (e_b - e_n)$$
(1.19)

To allow more general superlement representation of the constitutive law, the external force  $s_{nlb}$  can be defined defined by a first order equation involving the system strain state at a given Gauss point  $e_b$  and  $\dot{e}_b$ , and a series of internal states  $\mathbf{e}_i$  and  $\dot{\mathbf{e}}_i$ . In the following bold letters refer to non-scalar values for a given Gauss point.

$$\begin{bmatrix} K_{bb} & \mathbf{K}_{bi} \\ \mathbf{K}_{ib} & \mathbf{K}_{ii} \end{bmatrix} \begin{cases} e_b \\ \mathbf{e}_i \end{cases} + \begin{bmatrix} C_{bb} & \mathbf{C}_{bi} \\ \mathbf{C}_{ib} & \mathbf{C}_{\mathbf{ii}} \end{bmatrix} \begin{cases} \dot{e}_b \\ \dot{\mathbf{e}}_i \end{cases} = \begin{cases} s_{nlb} \\ \mathbf{0} \end{cases}$$
(1.20)

No external forces being applied on the internal rheology, one is able to use a Schur complement to obtain the internal states evolution equation

$$\dot{\mathbf{e}}_i = -\mathbf{C}_{ii}^{-1}\mathbf{K}_{ib}e_b - \mathbf{C}_{ii}^{-1}\mathbf{C}_{ib}\dot{e}_b - \mathbf{C}_{ii}^{-1}\mathbf{K}_{ii}\mathbf{e}_i$$
(1.21)

Once the internal state is resolved, the external force can be deduced

$$s_{nlb} = \begin{bmatrix} K_{bb} & \mathbf{K}_{bi} \end{bmatrix} \begin{cases} e_b \\ \mathbf{e}_i \end{cases} + \begin{bmatrix} C_{bb} & \mathbf{C}_{bi} \end{bmatrix} \begin{cases} \dot{e}_b \\ \dot{\mathbf{e}}_i \end{cases}$$
(1.22)

Resolution of equation (1.21) can be obtained with different strategies. In transient simulations a local integration is performed using the Euler scheme. From intstant  $t_n$  the internal state at instant  $t_{n+1}$  has to be integrated over  $h = t_{n+1} - t_n$ 

$$\mathbf{e}_i^{n+1} = \mathbf{e}_i^n + h(1-\theta)\dot{\mathbf{e}}_i^n + h\theta\dot{\mathbf{e}}_i^{n+1}$$
(1.23)

Implemented strategies can involve either single or multiple steps integrations in implicit or explicit form. Implicit resolution involes Newton-Raphson resolution, the usual initial prediction assuming constant velocity over the time step.

In the harmonic balance method, the global resolution scheme iterates on the internal states, so that one directly computes equations (1.21) and (1.22) from the current solution and the be used in the residue equation. The harmonic balance scheme then checks non-linear forces equilibrium and the stability of internal velocities.

#### 1.2.7 Iwan

Now let us consider a more non-linear constitutive law : the standard triboelastic solid.



Figure 1.3: STS model

The evolution equation is defined in a piecewise manner

$$\dot{x}_{k} = \frac{1}{K_{k} + K_{k+1}} (K_{k} \dot{x}_{k-1} + F_{k}) \quad if |F_{k}| > F_{f}$$
  
$$\dot{x}_{k} = 0 \qquad \qquad \text{otherwise}$$
(1.24)

with  $F_k = k_k(e_k - e_{k-1}) + k_{k+1}(e_k - e_{k+1})$ . When integrating, the algorithm is stopped as soon as a  $\dot{x}_k$  condition is reached.

# **1.3** Transient solution of non-linear equations

## 1.3.1 Principles

To allow the use SDT transients with external FEM packages, it is assumed that a superelement representation of the model is imported

$$[M_R] \{ \ddot{q}_R(t) \} + [C_R] \{ \dot{q}_R(t) \} + [K_R] \{ q_R(s) \} = [b_R] \{ u(t) \}$$
(1.25)

In general, the reduction is performed so that the DOFs retained  $\{q_R\}$  are related to the orignal DOFs of a larger model by a Rayleigh Ritz reduction basis T using

$$\{q\}_N = [T]_{N \times NR} \{q_R(s)\}_{NR}$$
(1.26)

This representation is fairly standard. The data structure representation within SDT is described in section 1.6.1 . SDT/FEMLink supports import from various FEM codes and more details are given in section 2.4 for NASTRAN, section 2.5 for Abaqus, and section 2.6 for ANSYS.

For transient resolution a real representation of damping must be used. Rayleigh and viscous damping are thus the only solutions supported. It is noted that for sine sweeps, it is possible to consider a time varying Rayleigh damping which has been found to be appropriate in some cases.

nl\_solve fe\_timeModalNewmark implements an optimized fixed time step version of the Newmark scheme (see [?] section 4.1.4) assuming a modal basis associated with the underlying linear system (discussed in section 1.3.2).

The non-linear resolution of the mechanical equation is usually performed by an iterative predictor/corrector scheme. Given the solution at time step n, the prediction is initialized by assuming a null acceleration at time step n + 1, so that the predictors  $q_{n+1}^0$  and  $\dot{q}_{n+1}^0$  are expressed as

$$\begin{cases} q_{n+1}^{0} = q_{n} + h\dot{q}_{n} + h^{2}(\frac{1}{2} - \beta)\ddot{q}_{n} \\ \dot{q}_{n+1}^{0} = \dot{q}_{n} + h(1 - \gamma)\ddot{q}_{n} \end{cases}$$
(1.27)

One considers the displacement correction  $\Delta q_{n+1}$  as the only unknown and velocity and acceleration at time step n + 1 are given by

$$\begin{cases} \Delta q_{n+1} = q_{n+1} - q_{n+1}^{0} \\ \dot{q}_{n+1} = \dot{q}_{n+1}^{0} + \frac{\gamma}{h\beta} \Delta q_{n+1} \\ \ddot{q}_{n+1} = \frac{1}{\beta h^{2}} \Delta q_{n+1} \end{cases}$$
(1.28)

Provided solution  $q_{n+1}^k$ , the residue is defined as

$$r_{n+1}^{k+1} = [M] \, \ddot{q}_{n+1}^k + [C] \, \dot{q}_{n+1}^k + [K] \, q_{n+1}^k - f_{cn+1} - f_{NLn+1}(q_{n+1}^k, \dot{q}_{n+1}^k, t_{n+1}) \tag{1.29}$$

and the correction is found by solving  $J\Delta q_{n+1}^{k+1} = r_{n+1}^{k+1}$  using the diagonal fixed Jacobian

$$J = \left[ \frac{1}{\beta h^2} + \frac{\gamma 2 \zeta_j \omega_j}{\beta h} + \omega_j^2 \right]$$
(1.30)

For one step formulation see [1] formula (4.53).

For a given system, a one-step Newmark is the combination of a linear evolution matrix depending on the linear system properties and time step h, and external forces. One thus writes the discrete state evolution equation as

$$\left\{\begin{array}{c}q_{n+1}\\\dot{q}_{n+1}\end{array}\right\} = [E(h)]\left\{\begin{array}{c}q_n\\\dot{q}_n\end{array}\right\} + \{f_{ch}\}\tag{1.31}$$

The evolution equation combines the quadrature rules and the mechanical equilibrium at states n and n + 1:

$$\begin{cases}
q_{n+1} = q_n + h\dot{q}_n + h^2\beta\ddot{q}_{n+1} + h^2\left(\frac{1}{2} - \beta\right)\ddot{q}_n \\
\dot{q}_{n+1} = \dot{q}_n + h\gamma\ddot{q}_{n+1} + h\left(1 - \gamma\right)\ddot{q}_n \\
M\ddot{q}_{n+1} + C\dot{q}_{n+1} + Kq_{n+1} = f_{cn+1} \\
M\ddot{q}_n + C\dot{q}_n + Kq_n = f_{cn}
\end{cases}$$
(1.32)

Multiplying the quadrature equations by M and replacing acceleration terms by their mechanical equation resolution provides the evolution equation that can be matricially written

$$\begin{bmatrix} M+h^{2}\beta K & h^{2}\beta C\\ h\gamma K & M+h\gamma C \end{bmatrix} \begin{cases} q_{n+1}\\ \dot{q}_{n+1} \end{cases} = \begin{bmatrix} M-h^{2}\left(\frac{1}{2}-\beta\right)K & hM-h^{2}\left(\frac{1}{2}-\beta\right)C\\ -h\left(1-\gamma\right)K & M-h\left(1-\gamma\right)C \end{bmatrix} \begin{cases} q_{n}\\ \dot{q}_{n} \end{cases} + \cdots \\ \begin{cases} h^{2}\beta f_{cn+1}+h^{2}\left(\frac{1}{2}-\beta\right)f_{cn}\\ h\gamma f_{cn+1}+h\left(1-\gamma\right)f_{cn} \end{cases} \end{cases}$$
(1.33)

The evolution matrix is then

$$[E(h)] = \begin{bmatrix} M + h^2 \beta K & h^2 \beta C \\ h\gamma K & M + h\gamma C \end{bmatrix}^{-1} \begin{bmatrix} M - h^2 \left(\frac{1}{2} - \beta\right) K & hM - h^2 \left(\frac{1}{2} - \beta\right) C \\ -h \left(1 - \gamma\right) K & M - h \left(1 - \gamma\right) C \end{bmatrix}$$
(1.34)

and the interpolated external force is then

$$\{f_{ch}\} = \begin{bmatrix} M + h^2 \beta K & h^2 \beta C \\ h\gamma K & M + h\gamma C \end{bmatrix}^{-1} \begin{cases} h^2 \beta f_{cn+1} + h^2 \left(\frac{1}{2} - \beta\right) f_{cn} \\ h\gamma f_{cn+1} + h \left(1 - \gamma\right) f_{cn} \end{cases}$$
(1.35)

The acceleration can then be resolved with one of the quadrature rules, the simplest being the velocity quadrature providing the relation

$$\ddot{q}_{n+1} = \frac{1}{h\gamma} \left( \dot{q}_{n+1} - \dot{q}_n \right) - \frac{1 - \gamma}{\gamma} \ddot{q}_n \tag{1.36}$$

## 1.3.2 Definition of an underlying linear system

When defining a non-linear constitutive law, it is always possible and often desirable to define an underlying linear system. Taking the simple case of a cubic spring where  $s_{nl} = e_{nl}^3$ . Figure 1.4 clearly illustrates the difference between the tangent stiffness, slope of force at current point  $3e_{nl}^2$ , and the secant stiffness, ratio of force divided by deformation  $e_{nl}^2$ .



Figure 1.4: Left : Tangent and secant stiffness. Right : possible underlying linear systems for the xxx example.

When defining a non-linear constitutive law, it useful and SDT-HBM requires that an underlying linear system be defined. For a general  $F_{nl}(e(t), \dot{e}(t))$  law, the non-linear stress used in time integration should thus be of the form

$$\{s_{nl}(t)\} = \{F_{nl}(e(t), \dot{e}(t))\} - [k_J] \{e_{nl}(t)\} - \{F_0\}$$
(1.37)

with  $[k_J] \{e_{nl}(t)\}$  the chosen linear representation of the non-linearity and  $F_0$  the value of the non-linear stress at the system state around which the response is computed.

When considering assembly in SDT, elements with a non-linearity defined through the NLdata field are ignored in linear assembly if NLdata.keepLin=0. For example, for a Maxwell model, reduction is best performed using the high frequency modulus. Thus a non-linear spring should be coupled with a linear spring using that high stiffness.

# 1.4 Data structures for non-linearities in time

#### 1.4.1 NLdata non-linearity definition

A non linearity has the following input form when stored as NLdata of an element group property.

- .type a MATLAB function handle which is used for initialization phases. The most commonly used non-linearity is nl\_inout. For transient simulations observation matrices are built in nl\_spring('init') while for HBM solutions this is done in hbm\_solve InitHBM.
- .Fu can be done through generic anonymous functions see section 1.2.4, user defined anonymous functions see section 1.2.3, tabular definitions section 1.2.2.
- .adof optional definition of DOF specific to the non linearity that will be propagated to model.FNLDOF, see below.

### 1.4. DATA STRUCTURES FOR NON-LINEARITIES IN TIME

- .isens optional : may be used to select a partial list of strains normally computed. For example to only keep translations of a cbush use .isens=[1 2 3].
- .adofi to declare internal states, in coherence with field .MatTyp below. This field can be omitted of left empty if no such feature is used. More details are given under DOF representation of internal states.
- .MatTyp : declares the time derivative of the signal associated to each internal DOF. When using explicit definitions of internal DOF, as in HBM or time integration with explicit internal states, this is used to specify how to compute the Jacobian. Thus MatType=1 stiffness coresponds to a displacement DOF, 2 mass an acceleration DOF, 3 viscous damping a velocity DOF.
- .obs Optional command to fill in out.FNL.NL, see below. This can either be a logical, evaluating nl\_fun('obs'), or a cell-array giving non linear data fields to propagate to the output from model.NL, or a string which will be evaluated.
- .snl as a empty field will force the initialization of a NL.snl buffer during time integration.
- .StoreType can be used to specify the StoreType strategy.

# 1.4.2 Additional fields in model

Base on model stack entries of the **pro** type where as **NLdata** field is present as described in the previous section, **nl\_spring** support additional model fields.

- .NL This is a stack containing all non linearity data built using nl\_spring NL. The stack has as many lines as existing non-linearities and three columns, the first being the non linearity type (the non linear function name most of the time), the second the non linearity name, and the non linearity data described in section 1.4.3.
- .FNL This is a transfer vector containing non linear data to output. Each non-linearity can store output data based on .Find.
- .FNLDOF This is a DOF vector corresponding to vector FNL. These FNLDOF are defined freely and not critical for base functionality. They are however convenient to keep track of internal states or non-linearity output, so that its building should be taken care of.
- .FNLlab This is a cell array of labels corresponding to non linear data to output. This is used by iiplot when non-linear responses are displayed during cleanup operations.

Output from simulations with non-linearities, from nl\_solve, or fe\_timehave additional data stored in a .FNL field. This is a deformation structure with fields

- .def Stored content of model.FNL at saving times.
- .DOF Stored content of model.FNLDOF.
- .data Copied data of out.data (saved times)
- .lab Stored content of model.FNLlab.
- .NL This is a cell array of the same format than model.NL. The third column contains structures containing non linear data relevant for result display. By default it contains field .iNL, the index vector of the current non linearity in the full FNLDOF. This field can be modified using the .obs callback of the non linearities. This field is based on NL.FInd, if .FInd is missing, iNL will be empty. iNL is then generated using the length of NL.adof, or by default using the length of NL.unl.

## 1.4.3 NL structure : non-linearity representation during time integration

During time integration, non-linearities are stored in the model.NL cell array. Each non-linearity is a data structure with the following standard fields for optimized computation (see mkl\_utils

- .type a MATLAB function handle which is then called through NL.type(NL,fc,model,u,v,q,opt,Case). Some older NL use a string giving the type name.
- .c observation matrix (1.2.5) for non linear displacements and velocities. During solves this is stored in row format obtained with NL.c=v\_handle('mklst',sparse(NL.c)) to optimize product speed.
- .unl pre-allocated memory for the result of NL.c\*u. Must be consistent with the number of rows in NL.c. The computation is handled by mkl\_utils. New implementations support a third dimension to store .unl0 in .unl(:,:,2) and .unlj1, the value at the previous time step, in .unl(:,:,3).
- .vnl if exists pre-allocated memory for the result of NL.c\*v. Must be consistent with the number of rows in NL.c. The computation is handled by mkl\_utils.
- .snl preallocated buffer of length size(b,2) to store the non-linear stresses (1.6).
- .b command matrix for non linear loads. At the end of the NL.type call it is expected that NL.snl (which may point xxx) contains the non linear component loads such that the residual becomes r=r+NL.b\*NL.snl.

During solves, sparse matrix operations must be optimized. This the product  $bs_{nl}$  is computed, it is interesting to store the transpose of the *b* matrix. The matrix is expected to be in

#### 1.5. HARMONIC BALANCE SOLUTIONS AND SOLVER

transposed form NL.b=v\_handle('mklst',b);. This conventions allow reuse of a .c matrix for command.

Note that the sign conventions when using unl to return a non linear force are opposite to what is done when the result is added to fc, see sdtweb nl\_fun to compare conventions.

- .FInd C++ start index in model.FNL to store the current non linearity data.
- .adof FNLDOF specific to the non linearity.
- .obs Optional command to fill in out.FNL.NL, see below. This can either be a logical, evaluating nl\_fun('obs'), or a cell-array giving non linear data fields to propagate to the output from model.NL, or a string which will be evaluated.
- .opt data vector containing *double* values to be used in C++ implementation of non-linearities. Default values would be FuCode(1) tc(2) dtO(3) val1 val2.
- .iopt int32 data vector containing non-linearity specific integers. (1)FInd start of model.FNL, (2)iu C staring position of internal states in global displacement vector, (3)Nstrain number of observed strains for each gauss point. (4)Nistate number of internal states per gauss point. (5)Ngauss number of gauss points in a vectorized non-linearity. (6)StoreType update strategy in residual call.

Optional for tabular (6)tStart, curStart C style start of table and cur in .opt. (8)size(Y,1), (9) dimensions of table with unl as first column and values in second.

- in the C implementations the N field gives [0]numel(unl(:,:,1)) [1]size(c,1) size(c,2) isTrans(c) [4]size(b,1) size(b,2) [6]isTrans(b) [8]length(opt) length(iopt) [10]size(b,2)
- obsolete
- .extDOF obsolete see iopt, double vector containing Matlab indices xxxGVxxx, [iu,Nstrain,Nistate]
- Obsolete .unl0 offset to apply to .unl, so that the content of .unl becomes NL.c\*u+unl0. *Current implementations* should store in .unl(:,:,2).
- Obsolete .vnl0 offset to apply to .vnl, so that the content of .vnl becomes NL.c\*v+vnl0. Current implementations should store in .vnl(:,:,2).

# 1.5 Harmonic balance solutions and solver

#### **1.5.1** Elastic representation as superelements

For interfacing with external finite element software the well documented superelement formalism is used. This formalism is largely used by Multibody Dynamic Software (Simpack, Adams, Excite, ...) and thus widely documented.

A superelement representation of the model is of the form

$$\left[M_R s^2 + C_R s + K_R + i D_R\right] \{q_R(s)\} = [b_R] \{u(s)\}$$
(1.38)

In general, the reduction is performed so that the DOFs retained  $\{q_R\}$  are related to the orignal DOFs of a larger model by a Rayleigh Ritz reduction basis T using

$$\{q\}_N = [T]_{N \times NR} \{q_R(s)\}_{NR}$$
(1.39)

This representation is fairly standard. The data structure representation within SDT is described in section 1.6.1 . SDT/FEMLink supports import from various FEM codes and more details are given in section 2.4 for NASTRAN and section 2.5 for Abaqus.

To allow the definition of elements within SDT, it is necessary to use superelements with physical DOF, typically known as MASTER degree of freedom. These are stored in SDT as a DofSet entry. Other columns of T may correspond to generalized or internal DOFs.

To use equivalent spring representations of volumes or surfaces, a first common approach is to connect a set of nodes two a single reference node. This kinematic link can be done as

- ridid assuming that the 3D motion of all nodes in the set depend rigidly on the center node motion. This tends to overstiffen the area of connected nodes.
- rbe3 assuming that the center node moves as the mean motion of the set of nodes. An appropriate weighting is generated as described xxx

These links exist in Abaqus ??, NASTRAN ??, ANSYS ??. Damping representation : G global or Rayleigh xxx link to proper SDT reference.

## 1.5.2 Time/frequency representation of solutions/loads

The solver is meant to support a generic representation of time dependence. The generalized degrees of freedom considered here are components of a vector noted Z and correspond to amplitudes multiplying functions of both space and time. For a scalar function of space (single spatial DOF), multiple harmonic DOF  $f_k$  and time varying shape functions  $h_k(t)$  fully describe the time dependence

$$\{f(t)\} = \sum_{k \in \mathcal{H}} f_k h_k(t) = \{f_k\}_{n_k}^T [H_{kt}]_{n_k \times n_t}$$
(1.40)

In the specific case of harmonic balance [2], the time dependence of a given DOF is written as

$$q(t) = z_{cq0} + \sum_{n \in \mathcal{H}} \left( z_{sqn} \sin\left(n\omega t\right) + B_{cqn} \cos\left(n\omega t\right) \right) = \sum_{k \in \mathcal{H}} Z_{qk} h_k(t)$$
(1.41)

where spatial DOFs are indexed by q and temporal DOFs are indexed by k but need also a representation of the harmonic n. Thus the ordering of generalized space/time DOFs is

$$\{Z_{qk}\} = \begin{cases} z_{cq0} \\ z_{sq1} \\ z_{cq1} \\ \vdots \\ \vdots \\ n_q.n_h \times 1 \end{cases}$$
(1.42)

with a spatial dependence q (degree of freedom in the time domain equation(1.1)) and a temporal dependence k. Note that in the notation above, the individual components are assumed real, but the problem can also be written in complex form (1.54). As always in SDT, it is expected that the ordering of Z could be changed. Thus a list of DOFs is needed to specify the meaning of each component of the vector. This list is given in the form

```
hdof={ % List of active DOFs
    1.01 'c0' % Node1.DOF1(x), constant(B0)
    10.02 's1' % Node10.DOF2(y), first harmonic (A1)
    10.02 'c1' % Node10.DOF2(y), first harmonic (B1)
};
```

where the first column specifies the spatial DOF and the label in the second column the temporal component. hbm\_solve harm commands provide utilities to manipulate spatio-temporal DOF definitions.

The definition of a solution is thus made using a data structure with fields

- .def matrix with columns being  $Z_{qk}$  values in given configuration (frequency dependence will be obtained with multiple columns).
- .hdof cell array giving the meaning of spatio-temporal amplitudes in .def rows (see equation (1.41))
- TR since, in general harmonic balance is performed using a reduced model, restitution of the full DOFs is based on with the reduction basis stored in def.TR.

Building of the time response of a given DOF q(t) should, using the convention of summed indices, actually be written as

$$q(t_j) = Z_{ql} H_{lj} \tag{1.43}$$

with  $H_{lj} = H_l(t_j)$  the  $l^{th}$  harmonic function at time  $t_j$ . Times are assumed to be sampled into the regular time interval  $[t_1 : t_N]$ , so that H can be written in this case as a matrix

$$H_{lj} = \begin{bmatrix} \mathbb{1}(t_1) & \dots & \mathbb{1}(t_N) \\ \sin(\omega t_1) & \dots & \sin(\omega t_N) \\ \cos(\omega t_1) & \dots & \cos(\omega t_N) \\ \vdots & \dots & \vdots \\ \sin(k\omega t_1) & \dots & \sin(k\omega t_N) \\ \cos(k\omega t_1) & \dots & \cos(k\omega t_N) \\ \vdots & \dots & \vdots \end{bmatrix}$$
(1.44)

In further equations and in the code, the notation  $Hkt(H_{kt})$  is used even though there are two *l* lines for each harmonic *k* (except for harmonic 0). xxx j

An advantage of the retained definition is that it is not necessary to define all harmonics. Furthermore problems with sub-harmonics of the excitation frequency can also be considered and only require development of appropriate label handling methods to define sub-harmonic functions  $h_k(t)$ .

The inverse transform is the way to obtain the harmonic amplitudes from a time vector and is associated with the Htk linear operator

$$Z_{ql} = \{q(t_j)\}_{Nq \times Nt} [H_{tk}]_{Nt \times Nt} = \{q(t_j)\} \frac{2}{N} \begin{bmatrix} \frac{1}{2} & \sin(\omega t_1) & \cos(\omega t_1) & \dots \\ \vdots & \vdots & \vdots & \dots \\ \frac{1}{2} & \sin(\omega t_N) & \cos(\omega t_N) & \dots \end{bmatrix}$$
(1.45)

It is thus verified that  $H_{kt}H_{tk} = [I]_{Nk \times Nk}$ .

Some of the literature considers linear DOFs (including modal DOFs and some physical DOFs), non-linear DOFs (internal or physical). SDT-HBM does not ask the user for the nature of DOFs since non-linear DOFs can actually be deduced from the consideration of DOFs present in non-linear observation matrices and the notion was not found to be needed.

The spatial DOF nature (physical, modal, internal) is also not necessary. It is however useful to have automated procedures to assign an identifier for every spatial DOF. It is thus expected that generalized (modal) DOFs be assigned a node number at the time of superelement creation. To avoid viewing mistakes DOF associated with modes or internal states are affected to DOF 99 (of the form NodeId.99).

Finally, some non-linearities use internal states. As the HBM solver needs explicit access to these states, the hbm\_solve InitHBM command performs a pre-processing step that affects a node number for each internal state so that they are present in q.

### 1.5.3 Harmonic balance equation (real DOF)

The base iterations are associated with a non-linear least squares minimization problem of the form

$$\min_{Z} \|\{R(Z,\omega,u)\}\| = \min_{Z} \|[A(\omega)]\{Z\} - [b]\{u_{ext}\} - [b_{nl}]\{s_{nl}(Z,\omega)\}\|$$
(1.46)

The contents of matrices  $[A(\omega)]$ , [b], and  $[b_{nl}]$  depend on the choice of time functions in (1.43). With the states described in (1.41) using (1.44), the rows of the residue matrix correspond to the work equilibrium equation (1.1) integrated over a period. Thus the harmonic 0 (constant term) is given by

$$R_0 = \sum_{i=1}^{N} (M\ddot{q} + C\dot{q} + Kq - F_{nl}(q, \dot{q}, q_{nl}, \omega, t_i) - F_{ext}(\omega, t_i))$$
(1.47)

The harmonic k leads to a sine (respectively cosine) contribution corresponding to an integral over the N time points of a period

$$R_{sk} = \sum_{i=1}^{N} \sin(k\omega t_i) (M\ddot{q}(t_i) + C\dot{q}(t_i) + Kq(t_i) - F_{nl}(q, \dot{q}, q_{nl}, \omega, t_i) - F_{ext}(\omega, t_i))$$
(1.48)

Assuming states ordered with sine contributions first followed by cosine, the generalized Jacobian matrix  $\frac{\partial R}{\partial Z}$  is composed of a linear part

$$A_{\mathcal{H}} = \begin{bmatrix} K & & \\ & K - (\omega)^2 M & -\omega D & \\ & \omega D & K - (\omega)^2 M & \\ & & \ddots & \\ & & K - (k\omega)^2 M & -k\omega D & \\ & & & K - (k\omega)^2 M & \\ & & & & \ddots \end{bmatrix}$$
(1.49)

and a series of non-linear contributions

$$J_{\mathcal{H}} = [b_{\mathcal{H}}] \frac{\partial \{s_{\mathcal{H}}(Z,\omega)\}}{\partial Z} = [b_{\mathcal{H}}] \frac{\partial \{s_{\mathcal{H}}(\epsilon_{\mathcal{H}},\omega)\}}{\partial \epsilon_{\mathcal{H}}} \frac{\partial \{\epsilon\}}{\partial Z} = [b_{\mathcal{H}}] \frac{\partial \{s_{\mathcal{H}}\}}{\partial \epsilon_{\mathcal{H}}} [c_{\mathcal{H}}]$$
(1.50)

The linear part of  $A_{\mathcal{H}}$  in (1.6.5) is skew-symmetric. The negative sign is applied to the lines corresponding to the sine contrutions. Using the notation (1.41), the negative sign is then located on the upper triangular part.

In the implementations derived from [?], the non-linear contributions are computed by finite differences (or possibly analytically although this is not yet implemented). And the inner loop iterations of the non-linear solver seeking the solution of (1.46) are of the form  $\Delta Z = [J]^{-1} R$ .

In direct frequency solvers by SDTools, one seeks to obtain the ideal single step convergence, where a sequant inter-harmonic stiffness  $[K_{\epsilon \mathcal{H}}]$  is found such that

$$\{s_{\mathcal{H}}\} = [K_{\epsilon\mathcal{H}}]\{\epsilon_{\mathcal{H}}\} \tag{1.51}$$

The two have the notable major difference that in general  $[K_{\epsilon \mathcal{H}}] \neq \frac{\partial \sigma}{\partial \epsilon}$ . In other words sequant and tangent stiffness are different matrices.

In the case of a scalar strain  $\epsilon$  and a linear complex stiffness of the form  $k(1+i\eta)$ . The interharmonic coupling is block diagonal and of the form

$$K_{\epsilon \mathcal{H}} = \begin{bmatrix} k & & & \\ & k & -k\eta & & \\ & k\eta & k & & \\ & & \ddots & & \\ & & & k & -k\eta & \\ & & & & k\eta & k & \\ & & & & & \ddots \end{bmatrix}$$
(1.52)

since  $-i\sigma_s + \sigma_c = k(1+i\eta)(i\epsilon_s + \epsilon_c) = ik(\epsilon_s + \eta\epsilon_c) + k(\epsilon_c - \eta\epsilon_s)$ . Anoter form considers an extended Z with frequency stored as an additional component (this is acheived using opt.Opt.fvar=1)

$$\min_{Z_e} \| [A_e(\omega)] \{ Z_e \} - b_e \|$$
(1.53)

In this case a last column  $\partial R/\partial \omega$  is added to the Jacobian. For the last row, the ideal would be to compute  $\frac{\partial \omega}{\partial Z}$  but the evaluation of this quantity is quite difficult, so that arc length techniques are used for continuation.

#### **1.5.4** Complex formulation and equivalent stiffness

To clarify the notion of equivalent stiffness found at the harmonic balance solution, the real harmonic balance states (1.41) can be written in complex form expressing the time response as

$$q(t) = \Re\left(\sum_{k \in \mathcal{H}} z_{qk} e^{ik\omega t}\right)$$
(1.54)

where spatial DOFs are indexed by q and temporal DOFs are indexed by k. Thus the ordering of generalized space/time complex DOFs

$$\{\mathcal{Z}_{qk}\} = \left\{\begin{array}{c} z_{q0} \\ z_{q1} \\ z_{q2} \\ \vdots \end{array}\right\}_{n_q.n_h \times 1}$$
(1.55)

The harmonic function is then complex,

$$H_{lj} = e^{il\omega t_j} \tag{1.56}$$

The usual harmonic balance using fourier coefficients, one has the equivalence

$$z_{qk} = z_{cqk} - iz_{sqk} \tag{1.57}$$

as  $e^{ik\omega t} = cos(k\omega t) + isin(k\omega t)$ , the equivalence between (1.54) and (1.41) is indeed

$$\Re(z_{qk})\cos(k\omega t) - \Im(z_{qk})\sin(k\omega t) = z_{cqk}\cos(k\omega t) + z_{sqk}\sin(k\omega t)$$
(1.58)

For each harmonic k the mean equilibrium over a period (1.48) leads to an equation of the form

$$\left(-(k\omega)^{2}M + ik\omega C + K\right)\left\{z_{qk}\right\} + \left(F_{nl}(q, \dot{q}, q_{nl}, \omega, t_{i}) - F_{ext}(\omega, t_{i})\right)\left[H_{tk}\right] = 0$$
(1.59)

where xxx

$$F_{nl}(Z,\omega) = -[b] s_{nl} \tag{1.60}$$

Since b is a constant matrix, the harmonic contribution of a non-linear stress can be computed at the gauss point level using complex notation

$$s_{k} = s_{ck} - is_{sk} = \mathcal{F}_{k}^{-1} \left( s_{nl}(Z, \omega, t_{i}) \right) = \sum_{i=1}^{N} \left( \cos(k\omega t_{i}) + i\sin(k\omega t_{i}) \right) \left( s_{nl}(Z, \omega, t_{i}) \right)$$
(1.61)

For a linear spring with a loss factor and harmonic strains  $u_k$ 

$$s_k = \mathcal{F}_k^{-1} \left( K(1+i\eta) u(Z,\omega,t) \right) = K(1+i\eta) u_k$$
(1.62)

Assuming a scalar stress relation, the non-linear harmonic problem (1.59) is thus equivalent to a linear harmonic problem

$$\left(-\omega^2 M + i\omega C + K + [b] K_{eq,k}[c]\right) \{\mathcal{Z}_k\} - F_{ext}(\omega) = 0$$
(1.63)

with the equivalent complex stiffness given for each gauss point by

$$K_{eq} = \frac{\mathcal{F}_k^{-1}\left(s_{nl}(Z,\omega,t_i)\right)}{u_k} \tag{1.64}$$

This expression of the problem is the basis for fixed point solvers, where a starting  $K_{eq,k}$  is introduced xxx

#### 1.5.5 Inter-harmonic coupling discussion

Harmonic balance Jacobian estimation based on local non-linear physics is never discussed in the literature. xxx

Possible for a given harmonic to estimate an equivalent local linear constitutive law. This notion comes close to quasi-Newtom methods in transient simulations, that estimate Jacobians with fixed operators based on the non-linear constitutive laws.

The harmonic balance framework adds some complexity as the transient response is decomposed in the time domain. It seems however possible to assess Jacobian relevant topologies using Taylor series expansion when the non-linear forces can be expressed in a functionnal way (coherent with the existence of a constitutive law).

$$[J_{nl}] = \frac{\partial f_{nlj}}{\partial z_{hqk}} \tag{1.65}$$

$$f_{nlj} \simeq k_1 \epsilon + k_2 \epsilon^2 + k_3 \epsilon^3 + \cdots$$
 (1.66)

$$\epsilon \simeq \epsilon_{cq0} + \epsilon_{cq1}\cos(\omega t) + \epsilon_{sq1}\sin(\omega t) + \epsilon_{cq2}\cos(2\omega t) + \epsilon_{sq2}\sin(2\omega t) + \epsilon_{cq3}\cos(3\omega t) + \epsilon_{sq3}\sin(3\omega t) + \cdots$$
(1.67)

First order terms induced by the linear and cubic constraint term respectively induced by  $\epsilon = \epsilon + \delta_{cq1} \cos(\omega t)$  and  $\epsilon = \epsilon + \delta_{sq1} \sin(\omega t)$ 

$$\begin{cases}
\left(k_1 - 9\epsilon_{cq1}^3 k_3\right) \delta_{cq1} \cos(\omega t) + \frac{3}{4} \epsilon_{cq1}^2 k_3 \delta_{cq1} \cos(3\omega t) \\
\left(k_1 + 9\epsilon_{sq1}^3 k_3\right) \delta_{sq1} \sin(\omega t) - \frac{3}{4} \epsilon_{sq1}^2 k_3 \delta_{sq1} \sin(3\omega t)
\end{cases}$$
(1.68)

First order terms induced by the linear and cubic constraint term respectively induced by  $\epsilon = \epsilon + \delta_{cq3} \cos(3\omega t)$  and  $\epsilon = \epsilon + \delta_{sq3} \sin(3\omega t)$ 

$$\begin{cases}
\left(k_{1} - \frac{9}{4}\epsilon_{cq3}^{2}k_{3}\right)\delta_{cq3}\cos(3\omega t) + \frac{3}{4}\epsilon_{cq3}^{2}k_{3}\delta_{cq3}\cos(9\omega t) \\
\left(k_{1} + \frac{9}{4}\epsilon_{sq3}^{2}k_{3}\right)\delta_{sq3}\sin(3\omega t) - \frac{3}{4}\epsilon_{sq3}^{2}k_{3}\delta_{sq3}\sin(9\omega t)
\end{cases}$$
(1.69)

First order terms induced by the linear and quadratic constraint term respectively induced by  $\epsilon = \epsilon + \delta_{cq1} \cos(\omega t)$  and  $\epsilon = \epsilon + \delta_{sq1} \sin(\omega t)$ 

$$\begin{cases} \epsilon_{cq1}k_2\delta_{cq1} + k_1\delta_{cq1}\cos(\omega t) + \epsilon_{cq1}k_2\delta_{cq1}\cos(2\omega t) \\ \epsilon_{sq1}k_2\delta_{sq1} + k_1\delta_{sq1}\sin(\omega t) - \epsilon_{sq1}k_2\delta_{sq1}\cos(2\omega t) \end{cases}$$
(1.70)
First order terms induced by the linear and quadratic constraint term respectively induced by  $\epsilon = \epsilon + \delta_{cq3} \cos(2\omega t)$  and  $\epsilon = \epsilon + \delta_{sq2} \sin(2\omega t)$ 

$$\begin{cases} \epsilon_{cq2}k_2\delta_{cq2} + k_1\delta_{cq2}\cos(2\omega t) + \epsilon_{cq2}k_2\delta_{cq2}\cos(4\omega t) \\ \epsilon_{sq2}k_2\delta_{sq2} + k_1\delta_{sq2}\sin(2\omega t) - \epsilon_{sq2}k_2\delta_{sq2}\cos(4\omega t) \end{cases}$$
(1.71)

#### 1.5.6 Load definition

The SDT definition of loads **DofLoad** or enforced displacement **DofSet** use the formalism of input shape matrices. Thus the time dependence of the load is given by

$$\{F(t)\} = [b] \{u(t)\}$$
(1.72)

where  $[b]_{N \times NS}$  describes the spatial content and the harmonic content is fully contained in  $\{u(t)\}$ . u is often scalar but can be a vector if multiple loads are combined. For each u component j, a two dimensional curve is defined giving

$$\{u_j(t)\} = u_{c0j}(\omega) + \sum_{k \in \mathcal{H}} u_{skj}(\omega) \sin(k\omega t) + u_{ckj}(\omega) \cos(k\omega t)$$
(1.73)

The harmonic load frequency dependence is then defined for each load vector  $b_j$  by scalar coefficients associated to each harmonic

$$\{u_{j}(\omega)\} = \begin{cases} u_{c0j}(\omega) \\ u_{s1j}(\omega) \\ u_{c1j}(\omega) \\ \vdots \end{cases}$$

$$(1.74)$$

These terms are declared by the field .curve defined in the Load structure.

- The field can be omitted or left empty. It then assumed that the force is a constant value associated to the c1 harmonic.
- The field can be a single curve, producing a scalar amplitude value per harmonic. The same amplitude is then applied to all loads.
- The field can be a cell array of curves, each curve being associated to a column of Load.def. The result of each curve is then coherent with the harmonics declared in the curve.

A given curve entry will provide the frequency dependency of a given Load vector for a specified set of harmonic shapes.

It can be defined using a *Tabular form*, or a *Functional* form by a structure coherent with sdtweb curve formats, with fields

- .X A 1x2 cell-array.
  - $.X{1}$ The first cell array provides the base pulsation vector that will provide the linear interpolation coefficients. This can be left empty for functional definitions.
  - $.X\{2\}$  The second cell is a column cell-array providing the harmonic shape labels to whom the load is applied.
- .Y the field providing the amplitude
  - Tabular form: a matrix with as many lines a in field  $.X{1}$  and as many columns a the number of harmonics provided in field  $.X{2}$ .
  - Functional form using MATLAB anonymous function handles. A structure with fields
    - \* .anonymous Provides the inline function. The anonymous function header is set by default if omitted, @(Zf,w). The inline can access the curve structure Zf whose fields will contain the fields declared in curve.Param, and the current pulsation w.
    - \* .csv A parameter declaration string under cingui ParamEdit format. This declares the parameters to be used, with a default value, their type and a possible brief explaination.
    - \* .Param The current parameters. .Param can be
      - a string defining the parameters declared in the .csv by par1=val1 par2=val2 ...
      - a structure with fields corresponding exactly to the declared parameters struct('par1' val1, 'par2', val2).

Any omitted parameter will be set to its default declared in the csv. Lack of default values would then results in an error at the function execution.

\* .tex a string providing a tex format of the formula used in .anonymous. Lazy declaration can be done by providing a string using w instead of the structure format.

```
% tabular formulation
C1=struct('X',{{1, ... % Frequencies can be a column vector if varying
        {'c0';'c1'}}}, ... % harmonic labels, see hdof
        'Xlab',{{'Frequency','harm'}}, ... % Frequency Hz or Pulsation rad/s
        'Y',[.1 50]); %
% Functional formulation
C2=struct('X',{{[],{'s1'}}},...
'Y',{{struct('anonymous','Zf.k1*w^2',...
'Param','k1=1e-2',....
'csv','k1(1#%g#"")',...
'tex','k_1 \omega^2')});
```

```
% Lazy anonymous
C3=struct('X',{{[],{'c0';'c2'}}},'Y',{{'10*sqrt(w)','w^2'}});
```

Internally this definition is transformed to use a command matrix  $[b]_{Nhdof \times Nhload}$ , with  $N_{hdof}$  the number of harmonic DOF defined in field .hdof and  $N_{hload}$  the number of harmonic loadings. One physical load is replicated by the number of harmonics specified in the curve field .X input to allow distinct amplitudes per harmonic.

At a given pulsation, a vector  $u(\omega)_{Nhload \times 1}$  is generated by parsing the curve inputs,

$$\{u(\omega)\} = \left\{ \begin{array}{c} \vdots \\ \{u_j(\omega)\} \\ \vdots \end{array} \right\}$$
(1.75)

so that the total external harmonic load vector is expressed as

$$\{Z_f\}_{Nhdof\times 1} = [b] \{u(\omega)\}$$
(1.76)

xxx Ease of use define c1a and c1phi xxx ?

# 1.6 Data structures for HBM solvers

#### 1.6.1 Model, superelement

The model structure containing in particular

- model.K list of matrices involved in the computation
- model.Klab list of labels describing each matrix
- model.Opt(2,:) list of labels describing each matrix
- model.NL stack of non-linearities.

This section describes a subset of superelement specifications described in more details in sdtweb('secms'). The structure is a standard OpenFEM model structure with additional fields described below. 38

Options characterizing the type of superelement as follows:

Opt(1,1)1 classical superelements.Opt(2,:)matrix types for the superelement matrices. Each non zero value on the<br/>second row of Opt specifies a matrix stored in the field K{i} (where i is<br/>the column number). The value of Opt(2,i) indicates the matrix type of<br/>K{i}. 1 stiffness, 2 mass, 3 viscous damping, 4 hysteretic damping.

#### Node

Nominal node matrix. Contains the nodes used by the unique superelement. 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.

#### 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.

#### 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).
- .hdof, 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

• .KeptDOF can be used to specify master DOFs not included TR.def but that should still be used for display of the superelement.

# 1.6.2 Non-linearity definition NLdata

Initialisation of non-linear behavior in a cbush element group is performed with the following NLdata formats xxx

Definition with custom functions. The NLdata property must contains the following fields

- type='nl\_inout' to let hbm\_solve InitHBM build the needed observation matrix
- Fu='@UserFun' references a user function computing the non-linear force with the prototype call described in section ??. Note that in instances of deployed MATLAB generated with the MATLAB compiler, all custom functions must be defined a priori. And only anonymous functions may be created.
- adofi to declare internal states, in coherence with field .MatTyp below. This field can be omitted of left empty if no such feature is used. Internal states are defined independently for each observation line used in the non-linearity. *e.g.* For a cbush six directions are available relative to the 3 translations and 3 rotations that can be observed. .adofi is then a line cell array of length the number of observations. Each cell defines a number of internal states associated to the corresponding observation index by providing a column vector with as many lines as internal states used each containing the DOF extension .99. The cell is left empty if no internal state is declared for a particular direction. NLdata.adofi={[];[];[];[];[];.99} will add an internal state to the 6<sup>th</sup> observation of the non-linearity.
- .MatTyp : declares the time derivative of the signal associated to each internal DOF. This corresponds to matrix definitions in the Jacobian. Thus MatType=1 stiffness corresponds to a displacement DOF, 2 mass an acceleration DOF, 3 viscous damping a velocity DOF.
- .isens may be used to select a partial list of strains normally computed. For example to only keep translations of a cbush use .isens=[1 2 3].

# 1.6.3 NL structure non-linearity representation during HBM solve

NL structures describing each non-linearity

- NL.c standard observation matrix for the observed motion used to express load. Initially in physical coordinates and transformed to harmonic observation in Build\_c\_unl\_k.
- NL.b command matrix to reapply harmonic loads on the proper DOFs.
- NL.Fu cell array of in-line functions.

- NL.type string containing the non-linearity type
- NL.c0,NL.b0 observation/command of non-linear strain in physical space only.

#### 1.6.4 Solver options definition

```
xxx
out=struct('Method', 'hbm_solve',...
 'Opt', [0 9 1 .01 .4 0 1], ... [adapt Nhmax nu fmin fmax UNU fvar]
 'SaveFreq', [0 .01 1e3 ], ... [stra (full/block) fstep nFpoints]
 'RelTol', 1e-9, 'MaxIter', 12, ...
 'Rayleigh',[0 0],...
 'NeedUNL',[0 0],...
 'AssembleCall', hbm_solve('AssembleCall'),...
 'JacobianUpdate', pmat(zeros(1)),...
 'iterHBM','@iterHBM',...
 'initHBM','@initHBM',...
 'resHBM','@resHBM',...
 'abscHBM', '@abscHBM',...
 'FinalCleanupFcn', 'hbm_solve(''fe_timeCleanup-cf-1'');',...
 ...'dSOpt', [ 2 1 0 .01 .01 .1 7 .3 1.5 6 12 ]); % [ Ldeg absc step dsfix dsmin dsma
 'dSOpt', [1 .01 .01 .1 7 .3 1.5 2 ]); %step(lin/lagrange) dsfix dsmin dsmax iopt bmin
```

- opt.Method ('hbm\_solve') Provides the method used by the solver. The default is hbm\_solve for a frequency scan.
- opt.AssembleCall (= hbm\_solve('AssembleCall'') Provides the call to perform an assembly performing initialization specific to the HBM module. This field is usually left by default, using the output of command hbm\_solveAssembleCall.
- opt.Jacobian (= '') Provides a callback to compute the Jacobian used for the solver resolutions. The default procedures peforms Jacobian computations during residue computation so that this field is empty by default.
- opt.JacobianUpdate (= pmat(ones(1))) A pmat indicator controlling the Jacobian updating procedure. This can be set to 0 to ask not to update the current Jacobian, or to 1 to trigger an update. This value can be modified by the solver if automated Jacobian update schemes are used (see opt.juit).
- opt.initHBM (='@initHBM') Provides a function handle called for data initialization before solve. The internal method initHBM is used by default, this field is thus initialized with the internal handle name of the hbm\_solve methods.

#### 1.6. DATA STRUCTURES FOR HBM SOLVERS

- opt.abscHBM (='@abscHBM') Provides a callback to perform curvilinear frequency predictions based on a buffer of response. The internal method abscHBM is used by default (provides linear or Lagrange polynom interpolation), this field is thus initialized with the internal handle name of the hbm\_solve methods.
- opt.iterHBM (='@iterHBM') Provides a callback to perform equilibrium iteration loops at a given point. The internal method iterHBM is used by default, this field is thus initialized with the internal handle name of the hbm\_solve methods.
- opt.resHBM (='@resHBM') Provides a callback to compute the equilibrium residue of a given HBM state. The internal method resHBM is used by default (also provides Jacobian computation), this field is thus initialized with the internal handle name of the hbm\_solve methods.
- opt.Opt (= [adapt Nhmax nu fmin fmax UNU fvar])
  - adapt (= 0) Unused, must be left to 0.
  - Nhmax (= 9) Defines the maximum number of harmonics considered, drives the transient buffer size. xxx Should directly be tBufSize, as convergence will depend on NL rather than Nh xxx
  - nu (= 1) Defines the harmonic factor  $\omega = k/\nu$  xxx.
  - fstart (= .01) Defines the starting frequency for scanning.
  - fend (= .4) Defines the end frequency for scanning.
  - UNU (= 0) Unused value left to zero for OpenFEM retro-compatibility.
  - fvar (= 1) Declares the frequency representation, either fixed (fvar = 0) or unknown (fvar = 1). If set to 1 the harmonics vector Z is augmented with the pulsation, corresponding to harmonic DOF 1.99, 'freq'.
- opt.Rayleigh (= [alpha beta]) Provides Rayleigh damping values applied to system matrices, defining  $C = \alpha M + \beta K$ .
- opt.juit (= -Inf) Defines an automated Jacobian update strategy used in resHBM, the Jacobian is then updated only after juit iterations. The default value set to -Inf asks for an update at each step.
- opt.dSOpt (= [stra dsfix dsmin dsmax iopt bmin bmax Ldeg]) Defines the curvilinear frequency prediction for continuation techniques, used by abscHBM.
  - stra (= 1) defines the increment strategy, either fixed (0) or with Lagrange polynoms (1).
  - fix (= .01 ) defines the fixed pulsation step , used for fixed strategy and to initialize other ones.

- $-\min(=.01)$  defines the minimum authorized pulsation step.
- $\max (= .1)$  defines the maximum authorized pulsation step.
- iopt (= 7) defines an optimal iteration number indicator. Pulsation frequency continuations will be modulated to target iopt iteration for convergence.
- bmin (= .3) defines the minimal modulation factor applied to reach iopt iterations for convergence.
- bmax (= 1.5) defines the maximal modulation factor applied to reach iopt iterations for convergence.
- Ldeg (= 2) defines the degree of the Lagrange polynomial extrapolation.
- opt.RelTol (= 1e-9) defines the tolerance to consider the HBM equilibrium is attained.
- opt.MaxIter (= 12) defines the maximum number of iterations allowed before stopping the loop.
- opt.SaveFreq (= [stra fstep nFpoints]) defines the output storage strategy.
  - stra (= 0) defines the storage strategy. 0 defines a direct output saving, saving as much as nFpoints results everytime the module frequency change is greater than fstep since the last save.
  - fstep (= .01) defines the frequency module evolution step triggering a save.
  - nFpoints (= 1e3) defines the total number of results saved in the output.
- opt.NeedUNL (= [0 0]) asks to save unl (if opt.NeedUNL(1)==1) and/or vnl (if NL.NeedUNL(2)== observations.
- opt.FinalCleanupFcn Provides the call to perform output cleanup after resolution. This field is usually left by default, calling hbm\_solve('fe\_timeCleanup''. Token -cf-1 directly stores the result in the GUI.

#### 1.6.5 Option structure during HBM solve

Inside the solver loop, developpers may want to access a number of parameters described below. The internal structure during time solves is described in sdtweb('nldata#nlformtime');

- opt.N number of samples for time signal
- opt.A A matrix (this matrix gets overwritten during iterations).
- opt.Hkt unit evolution of a given harmonic.(1.44)
- opt.dHkt time derivative of unit evolution of a given harmonic.

- opt.hdof two column matrix giving for each DOF the physical DOF and the time variation index. Transform to labels using xxx.
- opt.harm internal field of retained harmonics. opt.ihdof, opt.idof xxx

# 1.6.6 Harmonic result structure

is the data structure used to store SDT-HBM results. It is a variation of the **curve** format. With the first dimension containing harmonic DOFs, the second frequencies and the last amplitudes. xxx needs update xxx

```
sdtweb hbm_solve('outputinithbm')
out=struct('Y',zeros(r1),...
'X',{[{hdof}, freq,amp]},...
'Xlab',{'Hdof','Freq','Amp'},...
'hdof',{opt.hdof},'DOF',Case.DOF,...
'harm',opt.harm,'idof',opt.idof,'ihdof',opt.ihdof,'cur',zeros(1,max(3,length(r1)+1)))
```

# Tutorial

# Contents

2.1	Inst	allation	46
2.2	Free	quency domain test cases	<b>47</b>
	2.2.1	Example lists	47
	2.2.2	Spring mass examples	47
	2.2.3	Beam problem with local non-linearity	48
	2.2.4	Lap joint problem with contact	49
	2.2.5	Hyperelastic bushing	50
2.3	$\operatorname{Tim}$	e domain test cases	<b>50</b>
	2.3.1	Single mass test of various non-linearities	50
	2.3.2	Single mass stepped sine	51
	2.3.3	CBush with orientation	51
	2.3.4	Beam with non-linear rotation spring	52
	2.3.5	Lap joint with non-linear springs	52
2.4	NAS	STRAN interfacing	<b>52</b>
	2.4.1	Non-linear springs	53
	2.4.2	Superelement generation (Craig-Bampton type)	53
	2.4.3	Superelement generation (Free mode variants)	55
	2.4.4	Storing SDT-HBM options in bulk format	55
	2.4.5	Data export	57
2.5	AB	AQUS interfacing	<b>57</b>
	2.5.1	Non-linear springs	57
	2.5.2	Superelement generation (Craig-Bampton type)	57
2.6	ANS	SYS interfacing	<b>58</b>
	2.6.1	Non-linear springs	58
	2.6.2	Superelement generation (Craig-Bampton type)	58
2.7	Ava	nced usage	<b>59</b>
	2.7.1	DOE & general organisation in steps	59
	2.7.2	Obsolete NLVibKit	60

# 2.1 Installation

Steps of an installation are

- Install SDT.
  - The current starting point is SDT 6.8 beta which can be dowloaded from http://www.sdtools.com/distrib/beta/sdtcur\_dis.p. To obtain a SDT license key, you should then use the procedure at http://www.sdtools.com/faq/Release.html.
  - To save multiple SDT installations, see http://www.sdtools.com/faq/Release.html# multi
  - You must have write permission in the SDT directory so that it can be patched by yourself. If this is not the case you should install SDT somewhere in your own directories. For example use target='c:/sdtdata/hbm/sdt.cur' to avoid install to the traditionnal directory matlabroot/toolbox/sdt, see http://www.sdtools.com/faq/Release.html#multi.
  - For multi-boot systems (windows, linux), use a single SDT (see item above). This will avoid the need to install patches on both installations.
- For the target SDT **a patch is always needed**. Patches to SDT are files named hbm\_patch\_disp.p. They can be obtained using a call of the form

```
hbm_utils('DistribGetPatch') % Install the patch
hbm_utils('DistribCheck') % Run basic check of versions
```

- The command only downloads the patch, you are then supposed to click on the link cd('d:/del/scratch');hbm\_patch\_dis;rehash toolboxreset % do so that the patch is installed. The two step procedure is needed to give you a chance that the location of the patch install is correct.
- You must have write permission in the SDT directory, see first item of install.
- a copy of the SDT-HBM code. Two cases are possible
  - a deployement version obtained from SDTools as a crypted 7z file, this requires the existence of a license.txt file in your base SDT directory.
  - a SVN version obtained by a checkout on URL http://support.sdtools.com/svn/hbm/ trunk. For details on SVN clients to do a checkout, contact SDTools. It is expected that your rename your local copy of the trunk directory HBM. This will then be the base of your SDT-HBM directory structure.

The SDT-HBM directory structure is

#### 2.2. FREQUENCY DOMAIN TEST CASES

- hbm/m contains all the Matlab files needed to run SDT-HBM.
- hbm/help contains the documentation, see hbm\_utils Help to update.
- sdt.cur classical location for SDT installation associated with SDT HBM.
- hbm/test contains the files needed for testing. This is used for t\_hbm log.
- hbm/tex the root file is hbm.tex. It contains all the includes for other documentation files. Figures are stored as .pdf or .png in directory plots subdirectory, see hbm\_utils Latex to recompile. Documention contributions are welcome.

# 2.2 Frequency domain test cases

#### 2.2.1 Example lists

- d\_hbm('TestDuffing2dof') spring mass duffing example detailed in section 2.3.1
- model=d\_hbm('TestDofSet1') provides a test case with enforced displacement.
- d\_hbm('TestBeamNL') is a simple beam with a rotation spring. This was analyzed in detail in [3].
- d\_hbm('TestBeamVNL') is similar but implements the rotation spring as a single volume element. This is used to validate the implementation of volume non-linearities described section 1.1.5.
- d\_hbm('TestSPlate') illustrates a plate on non-linear supports.
- t\_hbm TestLapJoint Bj2 xxx provided by AGI

#### 2.2.2 Spring mass examples

The 2DOF duffing model provided by AGI is implemented in d\_hbm('TestDuffing2dof').



Figure 2.1: Two DOF duffing oscillator with cubic non-linearity

sdtweb d\_hbm('TestDuffing2Dof'); % Open source code of example
[mo1,opt,Z,XF]=d\_hbm('TestDuffing2dof'); % Run and display

#### 2.2.3 Beam problem with local non-linearity

#### sdtweb t\_hbm beamnl

As a first example clima16('BeamNLKrange'), one seeks to demonstrate the sensitivity to a variable stiffness. The non-linear stress strain relation is defined by

Fu=@(NL,fc,model,u,v,a,opt,Case,RO)kcur.\*(NL.unl-loss\*NL.vnl/opt.w);

which combines a constant stiffness kcur and a loss factor defined in the time domain using a frequency dependent velocity contribution. Taking  $q = cos(\omega t)$ , one assumes the equality of the complex stiffness and viscous damping forms in the stress/strain relationship  $s = Re(k(1 + i\eta)e^{i\omega t}) =$  $Re((k + i\omega ce^{i\omega t}))$  which leads to  $c = -\eta/\omega$ . In the resulting frequency responses below, one clearly sees a frequency response having a transition from a lower frequency at 4 Hz for kcur = 10e6 (in model units) and an upper frequency at 15 Hz for kcur = 1e8. The figure also clearly shows that damping decreases close to the limits as expected (see [3] for details).



As a second example clima16('BeamNARange'), one seeks to illustrate the amplitude dependence obtained for a stiffening spring. xxxDocument convergence issues.



#### 2.2.4 Lap joint problem with contact

Current simple test is found in t\_contact('LapJzt'). Variants include one or 3 bolts. Different strategies to generate the response.

The current test clima16('LJEB').



Figure 2.2: Lap joint with contact surface

# 2.2.5 Hyperelastic bushing

This example is a functional demonstration of capabilities associated with hyperviscoelastic behavior. It is based on the RotDamper example.

[mo1,hopt]=d\_hbm('TestBeamVNL');

# 2.3 Time domain test cases

• sdtweb('\_eval', 'd\_fetime.m#BumpStop') simple mass on spring with bumpstop non-linearity.

# 2.3.1 Single mass test of various non-linearities

Single mass test of various non-linearities sdtweb t\_nlspring ModalNewmark. Supported examples are

- Maxwell viscoelastic spring,
- tabular stiffness,
- Dahl model with constant normal force.

```
% Sample example with tabular stiffness and output spectrogram
sdtweb('_eval','d_fetime.m#Maxwell')
mdl=stack_set(mdl,'info','DefaultZeta',@(f)zeros(size(f)));
```

```
x=[-100 -1e-4 1e-4 100]';Fu=struct('X',{{x}},'Y',x.*[10 0 0 10]');
NLdata=struct('type','nl_inout','lab','TabK','keepLin',0,...
'isens',3,'MatTyp',{{[3]}},'Fu',{{Fu}});
mdl=feutil('setpro 1000',mdl,'NLdata',NLdata);
mo2=nl_solve('ReducFree 2 10 0 -float2 -SE',mdl);% With internal DOF
opt=nl_spring('TimeOpt dt=1e-4 tend=100 ModalNewmark');
RB=struct('spec','BufTime 20 Overlap .75 fmin0 fmax60 -window hanning','ci',3);
opt=stack_set(opt,'ExitFcn','Tip', ...
struct('FinalCleanup',{{'nl_solve','PostCdof'}},'DOF',2.03,'DoFreq',RB));
opt=stack_set(opt,'info','RangeTime',fe_range('grid',struct('A',logspace(-3,0,5))));
d2=fe_time(opt,mo2);d2=fe_def('subdef',d2,d2.data>d2.data(2));
```

# 2.3.2 Single mass stepped sine

The d\_hbm TestSteppedSine example illustrates the stepped sine procedure implemented with the ModalNewmark solver.

In such computations, it is assumed that the excitation frequency is fixed during the transient, so that the following parameters can be used the the run options

- .Nper number of periods for each computation
- .NperPer number of points per period. Alternatively time step .dt=1/freq/NperPer can be defined and used to set the initial value of .NperPer).
- .freq vector of frequencies for a series of stepped sine simulations.
- . A vector of amplitudes applied globally on the load case.

After the time computation of the target number of periods the ExitFcn entries in RT.Stack are processed by order. Typical entries would be

- PostFirstStab performs .ite time simulations without trying to check for stabilization to allow the transient to stabilize before any
- PostConstit extract non-linear stress/strains and possibly performs harmonic extraction with doFreq callback.

## 2.3.3 CBush with orientation

The example is detailed in

d\_fetime('tutoCbushOrient')

#### 2.3.4 Beam with non-linear rotation spring

First do a sweep

```
sdtweb('_eval','d_fetime.m#TestBeamNLRed')
%xxx
```

Now a drop from initial velocity

#### 2.3.5 Lap joint with non-linear springs

This test case is discussed with *Marco Rosatello*. See t\_bjoint('TestTime').

The axial behavior of the C2 connector is given by a non-linear law in tabular  $F_z(e_3)$  form. This force is output as the generalized stress during time computations, while for equations actually solved one uses

$$s_3(t) = F_z(e_3(t)) - k_{Jz}e_3(t) - F_0$$
(2.1)

For two directions x and y, the tangential behavior is incremented using a Dahl integration scheme

$$F_x(t+dt) = F_x(t) + \sigma(e_3(t))dt \ \dot{e}_x \left(1 - \frac{F_x(t)}{\mu F_z(e_3(t))} \text{sign}(\dot{e}_x)\right)^{\alpha}$$
(2.2)

Since this integration does not guarantee  $|F_x(t)| \leq |\mu F_z(t)|$ , the condition is enforced at each time step. Again there is the need to distinguish  $F_x(t)$  and the tangential load which needs to account for adherence stiffness in the nominal model used to generate the modal basis serving to define DOFs for time integration. Thus

$$s_1(t) = F_x(t) - k_{Jx}e_1(t) \tag{2.3}$$

Iwan model with Dahl cells. Data is sigma slope at no load,  $\alpha$  shape parameter,  $\mu F_z$  normal load.

$$F_x(t+dt) = F_x(t) + \sigma dt \ \dot{e}_x \left(1 - \frac{F_x(t)}{\mu F_z} \operatorname{sign}\left(\dot{e}_x\right)\right)^{\alpha}$$
(2.4)

# 2.4 NASTRAN interfacing

For generalities about superelements see section 1.5.1 . This section discusses the specifics of superelement generation using NASTRAN.

# 2.4.1 Non-linear springs

For spring representations of volumes or surfaces, a first common approach is to use so called rigid elements. NASTRAN supports

- RBE2 : rigid connections where the spring is connected to a master node with 6 DOF which enforce motion of a number of slave DOFs. xxx Example
- RBE3 : flexible connection where the spring is is connected to a slave node with 3 or DOF which depend from a set of master nodes. Note that in RBE3, the normal operation is to define a single slave node. UM cards in the RBE3 element can be used to change this behavior by selecting which DOF get eliminated.

Laws without internal states are similar to PGAP and import will be implemented in the future.

# 2.4.2 Superelement generation (Craig-Bampton type)

The superelement generation by NASTRAN is saved to an .op2 file that is automatically transformed to the SDT superelement format by FEMLink. A sample file is given in ubeamse.dat.

```
ASSIGN OUTPUT2='./ubeam_se.op2',UNIT=30
$
ID DFR
SOL 101
GEOMCHECK NONE
TIME 100
$
CEND
TITLE=Generic computation of mode shapes
METHOD=1
DISP(PLOT) = ALL
SPCFORCES(PLOT)=ALL
MPCFORCES(PLOT)=ALL
$ Now extract stresses on base
SET 101=1 THRU 16
STRESS(SORT)=101
$
MPC=1
SPC=1
$ RESVEC(NOINRL) = YES
EXTSEOUT (ASMBULK, EXTBULK, EXTID=100, DMIGOP2=30)
PARAM, POST, -2
```

```
PARAM, BAILOUT, -1
$
BEGIN BULK
$EIGRL,SID,V1,V2,ND,MSGLV,MAXSET,SHFSCL,NORM
EIGRL, 1, , , 20
$ DOF and nodes to support modal DOF
QSET1,0,1000001,THRU,1000050
SPOINT, 1000001, THRU, 1000050
$ Master DOF 4 base corners
BSET1,123,1,5,8,12
$
$ Residual on 3 DOF of input node 104
USET, U6, 104, 123
$
$ Residual associated with CAMP1 vector
CDAMP1
                 2
                         114
                                  1
                                           244
                                                   1
        161
PDAMP*
        2
                         1.
$
include 'ubeam_include.bdf'
ENDDATA
```

The resulting basis has the following form

$$[T] = \begin{bmatrix} I & 0 & 0\\ -K_{CC}^{-1}K_{CI} & [\phi_C]_{1:NM} & \left[K_{CC}^{-1} [b_{res}]\right]_{\perp} \end{bmatrix}$$
(2.5)

The op2 file contains nodes and superelement definition. It is advised to read the bulk file to obtain a model containing elements and material properties

- The interface DOF are defined in NASTRAN usingBset cards. These are stored in SDT as a DofSet entry to the model.
- QSET correspond to modal/generalized DOFs. These require the definition of a QSET card (to declare existing DOFs), SPOINT grids (to have node numbers to support these QSET DOFs). Note also that the SPOINT numbers should be distinct from other NodeId. The number of modes defined in the EIRGL card should be lower than the the number of SPOINT and the QSET card.
- Fixed interface modes  $\phi_c$  are computed by specifying the EIRGL card.
- Residual loads  $b_r es$  are defined as follows and lead to additional shapes using the residual vector procedures of NASTRAN

54

#### 2.4. NASTRAN INTERFACING

- point loads simply declared using the USET, U6 card
- relative loads simply obtained by declaring a CDAMP element that generates a relative viscous load between its two nodes.

[b] is defined by an DAREA real loading and possibly DPHASE definition. It should be noted that in SDT, it is strongly advised to define the phase using the input, since a complex input shape matrix has no sense in the time domain. The input is defined using a RLOAD2  $B(f)e^{i\phi(f)+\theta-2\pi f\tau}$  or RLOAD1  $(C(f)+iD(f))e^{i\theta-2\pi f\tau}$ 

#### 2.4.3 Superelement generation (Free mode variants)

When using a free mode computation, NASTRAN provides mechanisms to compute residual vectors, you should just insert the RESVEC=YES card. An example is given in the ubeamfr.dat file. The resulting basis has the following form

$$[T] = \left[ \left[ \phi \right]_{1:NM} \quad \left[ K_{Flex}^{-1} \left[ b_{res} \right] \right]_{\perp} \right]$$
(2.6)

The main mechanisms to generate residual vectors are

- Free interface modes  $\phi_c$  are computed by specifying the EIRGL card.
- Residual loads  $b_r es$  are defined as follows and lead to additional shapes using the residual vector procedures of NASTRAN
  - point loads simply declared using the USET, U6, NodeId, DofList card. Alternatively RVDOF (MSC but possibly not NX-NASTRAN) can given a list of up to four NodeId, Dof per card.
  - relative loads simply obtained by declaring a CDAMP element that generates a relative viscous load between its two nodes.

#### 2.4.4 Storing SDT-HBM options in bulk format

SDT-HBM provides an harmonic definition mechanism (see hdof). Storage in NASTRAN bulk format is as follows

\$	1	\$\$	2	\$\$	3	\$\$	4	\$\$	5	\$\$	6	\$\$	7	\$\$	8	\$\$	9	\$
\$	A11	DOFs	wit	h sir	ı(om	ega	t) a	nd co	os(on	nega	t)							
D	ΓI	HDO	)F	1		AL	L	123	3456	CS:	1	ENI	DREC					
\$	Grad	dual b	ouil	ding	of	full	lis	t of	DOFs	5								
D	ΓI	HDC	JF	1		N1		TH	RU	N2		123	3	S1		N3		
		THE	RU	N4		1		C1		N5		123	3456	S1		ENI	DREC	

Node numbers are first specified using ALL all (independent) nodes, N1 THRU N2 a list of consecutive node numbers, N5 a single node number. Associated DOFs are then written using the CM field of RBE2 (Component numbers of the dependent degrees-of-freedom integers 1 through 6 with no embedded blanks). A third field then specifies the harmonics. cs1 is a short cut for both  $cos(1\omega t)$  and  $sin(1\omega t)$ .

The specification of target frequencies follows the normal NASTRAN format using FREQ or FREQ1 cards. Provision for a single call generating responses at multiple amplitudes (hbm\_solve AFMap .Freq and .Amp fields) is specified as a DTI HBMAmp entry with all target amplitudes given.

\$ 1 2 \$\$ \$\$ \$\$ 5 \$\$ \$\$ \$\$ \$\$ \$ \$\$ 3 4 6 7 8 9 EIGRL, 10, , , 1 \$FREQ,SID,F2,F2,F3 \$FREQ1,SID,F1,DF,NDF FREQ 10 0.318 1. 3.0 4.0 RLOAD1 10 1 1 \$ 1 2 \$\$ 3 \$\$ 4 \$\$ 6 \$\$ 9 \$\$ \$\$ 5 7 \$\$ 8 \$\$ \$ DTI HBMAmp 1 1.0 2.0 3.0 ENDREC

To specify loads, a number of formats are defined.

\$	1	\$\$	2	\$\$	3	\$\$	4	\$\$	5	\$\$	6	\$\$	7	\$\$	8	\$\$	9	\$
DTI		Name		1	1 SID		)	101		FOF	FORM	Amp		UN1		UN2		
		Har	mi	ACi		ASi	-	ENI	DREC									

- Name is an arbitrary string (at most 8 characters) but should be unique and differ from internal nastran tables. By default it is proposed to use strings of the form P101 where 101 is the property number. But a more meaningful name might be relevant xxx. xxx future database xxx
- IREC (field 3 of the DTI) is only used when considering multiple entries with the same name and should be set to 1.
- SID : first the string SID in field 4 then, in field 5, the property identifier (integer) which should correspond to the set identification number SID for which this amplitude dependence is defined.
- Form (selected with the string on field 7) is the form name with the following formats defined

Form		
Amp	amplitudes	$\{u(t)\} = C_0 + \sum_{k \in \mathcal{H}} S_k sin(k\omega t) + C_k cos(k\omega t)$
AmpT	Amp table	$\{u(t)\} = C_0(\omega) + \sum_{k \in \mathcal{H}} S_k(\omega) \sin(k\omega t) + C_k(\omega) \cos(k\omega t)$

#### 2.5. ABAQUS INTERFACING

- Harmi number of retained harmonic. 1 for  $\cos(1\omega t)$  and  $\sin(1\omega t)$ .
- ACi, ASi amplitudes associated with the cosine and sine harmonic contributions. In the AmpT form integer numbers refering to table entries in the bulk.

# 2.4.5 Data export

nas2up supports writing of field at DOF results op2 format.

# 2.5 ABAQUS interfacing

For generalities about superelements see section 1.5.1. This section discusses the specifics of superelement generation using ABAQUS.

# 2.5.1 Non-linear springs

For spring representations of volumes or surfaces, a first common approach is to use so called rigid elements.

Abaqus supports several types

- **\*KINEMATIC COUPLING** : rigid connections where the spring is connected to a master node with 6 DOF which enforce motion of a number of slave DOFs. xxx Example
- \*DISTRIBUTING COUPLING (RBE3) : flexible connection where the spring is is connected to a slave node with 3 or DOF which depend from a set of master nodes.
- **\*COUPLING** : specific surface based definition, followed by either a **\*KINEMATIC** card for rigid or **\*DISTRIBUTING** card for RBE3 formulations.
- \*MPC : node based definition with type BEAM to constraint 6 DOF per node or type PIN to constraint the 3 translations only.
- **\*CONNECTOR** : connectors provide advanced structural kinematics, type **BEAM** without elasticity definition provides a rigid connection (linearized in SDT).
- **\*EQUATION** : generalized MPC definition with a direct constraint matrix declaration.

# 2.5.2 Superelement generation (Craig-Bampton type)

Superelement generation in Abaque is divided in three steps.

• \*STEP, PERTURBATION//\*STATIC used to define residual vectors. Note that export of residual loads associated with non-linearities is not yet implemented in SDT.

- **\*FREQUENCY**, **EIGEN=LANC** to compute internal modes with possibly a Craig-Bampton interface declared by a **\*BOUNDARY** card.
- \*SUBSTRUCTURE GENERATE to generate and export the superelement, use \*RETAINED NODAL DOF associated to fixed DOF in the frequency step for a Craig Bampton reduction.

See SeGenResidual.inp

# 2.6 ANSYS interfacing

#### 2.6.1 Non-linear springs

For spring representations of volumes or surfaces, a first common approach is to use so called rigid elements. ANSYS supports

- CE, CERIG, MPC184, RBE 2 : rigid connections where the spring is connected to a master node with 6 DOF which enforce motion of a number of slave DOFs.
- TARGE 170+CONTA 173, TARGE 170+CONTA 174

## 2.6.2 Superelement generation (Craig-Bampton type)

The cards typically used for superelement generation in ANSYS are

- antype, substr specify FE substructure generation in after /SOLU
- cmsopt,fix,Nshapes,,,,,tcms card generates .sub(import with ans2sdt('read','file.sub')) and .tcms files.
- resvec, on to use residual vectors in the basis
- seopt, *name*, MatType, 1 with MatType=2 for mass and stiffness, and 3 for stiffness, mass, viscous damping, *name* must be defined with card /FILENAME before the analysis.
- m, *NodeId*, all master DOF definition repeat card for the various interface nodes. You will have to replace all with UX,, UY, UZ, ROTX, ROTY, ROTZ if the DOF is used by an element that supports multi-physics.

```
/FILENAME, ubeam_se ! name must be the one used in SeOpt command
/PREP7
1...
! Use command F to apply loads that will define the residual vector
/SOLU
antype, substr ! substructure analysis
CmsOpt, Fix, 20, , , , , TCMS
RESVEC, ON
SeOpt, ubeamse_ans, 3, 1, 0,, ! 3(all matrices, 2 for m and k), 1 to print
! Define list of master DOF, you cannot use ALL if the elements support multi-physics
M, 1, UX, ,, UY, UZ, ROTX, ROTY, ROTZ
M, 5, UX, ,, UY, UZ, ROTX, ROTY, ROTZ
M,8,UX,,,UY,UZ,ROTX,ROTY,ROTZ
M, 12, UX, ,, UY, UZ, ROTX, ROTY, ROTZ
SAVE ! save .db file
SOLVE ! generate the matrices
FINISH
```

# 2.7 Avanced usage

## 2.7.1 DOE & general organisation in steps

The SDT fe\_range architecture supports the generic definition of numerical experiments.

- 10 Import linear model (SDT/OpenFEM format)
- 20 Define non-linear properties NLdata for a group of elements (non-linear springs CBUSH, contact surfaces, zero thickness element, StressCut)
- 30 Define computational range and options
- 40 Solve and save results
- 50 Post-process automatically

```
model = % SDT/OpenFEM format
     Node: [2x7 double]
      Elt: [6x9 double]
    Stack: {2x3 cell}
model.Stack = % List of properties
   % Case : Store boundary conditions and load
                             [1x1 struct]
   'case'
            'Case 1'
   % Pro : properties of a group of non-linear elements
    'pro'
             'nl_pro2001' [1x1 struct]
NL=stack_get(model,'', 'nl_pro2001', 'get')
      type: 'p_spring'
        il: [2001 2.2503e-02] % Usual elastic properties
    NLdata: [1x1 struct]
NL.NLdata= % Input format for non linearity data (CLIMA-HBM)
        type: 'nl_inout'
          Fu: '@(x)-.01*x.^3'
        Sens: 'cubicSpring'
     keepLin: 0
```

#### 2.7.2 Obsolete NLVibKit

In NL\_VibKit, the information stored in **adof** can be used to build the data in ZDEF, HARMDEF, VALDEF, BLDEF.

Method 1 : amplitudes

$$\{u(t)\} = C_0 + \sum_{k \in \mathcal{H}} S_k sin(k\omega t) + C_k cos(k\omega t)$$
(2.7)

C0,S1,C1, ...

Method 2 amplitude and phase

$$\{u(t)\} = C_0 + \sum_{k \in \mathcal{H}} S_k sin(\frac{k}{v}\omega t + \phi_k^s) + C_k cos(\frac{k}{v}\omega t + \phi_k^C)$$
(2.8)

 $C0,S1,phis1,C1,phic1, \dots$ 

Method 3 frequency dependent amplitude (associates a table to each)

$$\{u(t)\} = C_0(\omega) + \sum_{k \in \mathcal{H}} S_k(\omega) \sin(k\omega t) + C_k(\omega) \cos(k\omega t)$$
(2.9)

C0,S1Id,C1Id,  $\dots$ 

Method 4 : constant power (frequency dependendt loads like unbalanced loads)

$$\{u(t)\} = C_0 + \omega^n \sum_{k \in \mathcal{H}} S_k \sin(\frac{k}{v}\omega t) + C_k \cos(\frac{k}{v}\omega t)$$
(2.10)

n,C0,S1,C1,S2,C2, BLDEF, ACCDEF, VELDEF, DISPDEF

$$\{\ddot{x}_{23}\} = T_{1001}(\omega)sin(\omega t) \tag{2.11}$$

ZDEF, 23, p, lin, 784, 3, 66 HARMDEF, 66, 166, 1 VALDEF, 166, 0 0 ACCDEF, 3, 23, 1001 0 TABLEDEF,1001,10 1e-5 100 200 A1 : Tabledef,1001,10,1e-5,100,200

$$\{\dot{x}_{36}\} = 0.1sin(\omega t) + 0.74sin(5\omega t) \tag{2.12}$$

ZDEF, 36, p, lin, 159, 1, 10 HARMDEF, 10, 100, 1, 3, 5 VALDEF, 100, 0 0, 0 0, 0 0 VELODEF, 1, 36, 0.1 0, 0 0, 0.74 0

$$\{\dot{x}_{15}\} = 1.47 \sin(\omega t + \frac{p_i}{4}) + 0.5 \sin(\omega t + \frac{p_i}{4})$$
(2.13)

ZDEF, 15, p, nl, 83, 4, 3 HARMDEF, 3, 30, 1, 2 VALDEF, 30, 0 0, 0 0 DISPDEF, 2, 15, 1.47 0.7874 0.5 0.7874, 0 0 0 0

$$\{\ddot{x}_{12}\} = 5.8\omega^2 \cos(2\omega t) \tag{2.14}$$

ZDEF, 12, p, nl, 6, 1, 1 HARMDEF, 1, 1, 1, 2 VALDEF, 1, 0 0, 0 0 ACCDEF, 4, 12, 5.8, 2, 0 0, 0 1

# **Function reference**

# Contents

d hhm		65
	•••••	00
hbmui		66
$hbm_post$		73
hbm_solve		77
$hbm_utils$		85
nl_spring		87
$mkl_utils$		99
chandle	1	.02
Non linearities list	1	.04
Creating a new non linearity: nl_fu	n.m 1	.19
nl_solve	1	.22
nl_mesh	$\_\dots\dots$ 1	.30
$spfmex_utils$	1	.37

LIST OF THE SDT-HBM MODULE FUNCTIONS					
Function	Description				
d_hbm	Open source examples				
hbm_solve	Solving tools				
$\mathtt{hbm\_post}$	Post process tools				
hbmui	User interface				
hbm_utils	Maintainance				

# d\_hbm \_\_\_\_\_

## Purpose

Open source examples for HBM

# $\mathbf{Syntax}$

 $d\_hbm$ 

# TestBeamNL

Simple bending beam with localized non-linearity.

model=d\_hbm('testBoltedJoint')

# hbmui

# Purpose

Graphical user interface for the HBM solver (requires SDT)

# Syntax

#### hbmui

# Description

hbmui operates the GUI for HBM simulation procedures including pre, post treatment and simulation runs.

# Commands

# Hide

Not do display the GUI while running the HBM module.

hbmui('hide');

By default the GUI gets opened when the module is first loaded. Use this as the first call to the module to prevent the GUI from appearing. The GUI will remain hidden until an explicit call to hbmui is performed.

# Init[,Project,Post,...]

Initializes the GUI and specific tabs. If the tab already exists, the display will be switched to the existing data.

hbmui init hbmui initProject

The following command options are supported

- -Reset To reset the full GUI before the asked initialization.
- -resetCurTab To reset the tab specified for initialization.
- -noTab To initialize tab data without actually displaying it.

# PARAM[.Tab,UI]

Access to GUI parameter values.

#### RO=hbmui('PARAM.'Tab')'

The output is a struct RO with fields corresponding to the parameter names in the tab *Tab*, and values interpreted from the current GUI state. By default an error will be issued if the mentionned tab has not been initialized. The following command options are supported

• -safe Not to generate an error if the tab has not been initialized, but rather return the default values.

• -r1j To recover the parameter underlying java object.

• .*Par* To recover parameter named *Par* only.

Command PARAMUI returns the complete application UI structure.

#### Set[Project,Post,...]

Script version of GUI parameters sets.

hbmui('setTab', struct('Par', 'Val,...));.

Tab is the tab name containing the parameter to set, *Par* is the parameter name, *Val* is the value assigned to the parameter. To recover parameters names, see hbmui PARAM. To trigger an action linked to a push button, the value do must be assigned, *e.g.* hbmui('setPost',struct('refresh','do'));

Project



Figure 3.1: Project tab in initial state

#### Post

The **Post** tab handles post-treatment procedures allowing data export and display. In display mode it is linked to an HBM result object whose state can be altered through the GUI. To be used results must have been stored in the application.

The initial **Post** tab is presented figure 3.2,

#### hbmui

Eile Desktop	<u>W</u> indow <u>H</u> elp □¶   <del>∨</del>			r								
🕑 🖂 🛸 🛄	Post )											
Project	PostEnv											
Post	Load results											
	Results list		Pick John									
	itesuits list		Tick jobs									
	Layout	Overlay		-								
	PostFigure	11.0										
	PostSens											
	Sens	Type	Harm									
	+ Pick Obs	type	- Horni									
	PostSensList		Set harm for all									
	Plot Commands											
	_nTSamp	1000.0	1000.0									
	-Transfer plot											
	-Response	Synthetiz	Synthetized									
	E-Freq plot											
	LTHD plot		THD									
	E-Time plot											
	-Freq point	0.0										
	-Animate		9_									
	-Harm plot											
	Reset		💋 Refresh									
	Print 🗃 Print		📣 SaveFig									

Figure 3.2: Post tab in initial state

It features three sections

- **PostEnv** Handles the post-treatment environment, namely the selected job, with the possibility to load one, the **iiplot** figure number to host display. xxx Layout unsure
- PostSens Handles the observation stack, PostSensList. The first two lines allow handling the following ones that correspond to the actual observations. The first line proposes an interactive definition of a new post-treatment, see hbmui PostDlgSensPick. The second one is header to the list, with the possibility to apply changes to all following list entries regarding harmonic selection Set harm for all, selection toggle or full observation list deletion. The table features 5 columns
  - Sens Provides the observation set label.
  - Type Provides the type of signal, either disp, vel or acc for respectively a displacement, velocity, or acceleration signal.

- Harm Provides the harmonics retained for the post-treatment. The input will be treated as a subset of the job output harmonics. It can set either as a comma separated integer list, or a token (all, even, odd being supported).
- Out To toggle enabling of the current observation. If unchecked the observation list will be ignored in the post-treatments.
- In To select the current line as an entry for transfer displays. A maximum of one observation line can be selected as an input.
- R To suppress the corresponding observation line.
- Post Commands Drives the display options and commands. The following parameters can be set (presented in the format *Tab.Par*)
  - Post.RespSyn The type of synthesis to be performed, either Synthetized to synthetize the response on observation with all retained harmonics, byHarm to generate an observation response per harmonic, or byShape to generate an observation response per time shape.
  - Post.nTSamp Provides the number of time samples for transient displays.
  - Post.FreqPlot To select a maximum amplitude response as function of the frequency plot.
  - Post.TimePlot To select a transient response plot at a given frequency.
  - Post.FreqPoint To provide a frequency point associated to the transient response plot.
  - Post.HarmPlot To select a harmonic contribution bar graph at a given frequency xxx.
  - Post.TransferPlot To generate a transfer response between two observations.

The following commands can be launched

- Post.PlotReset To reset the iiplot figure with the current setup.
- Post.PlotRefresh To resfresh the display with the current setup.
- Post.PlotPrint To print the display into a plot, support for automatic reporting is provided throught the Porject tab setup.
- Post.PlotSaveFig To save a figure containing the current display.
- Post.THDPlot xxx
- Post.TimeAnim To animate the transient response associated to the current setup.
#### PostDlgSensPick

The SensPick dialog box provides tools for an interactive definition of observations based on jobs stored in the application. When clicking on the PickObs button in the PostSens section of the hbmui Post tab, the dialog box presented in figure 3.3 opens.

-SensName -Base job 200 -Sensor set -Pick sens set -Feature -Select in feature	Pick SensDof
Base job	Pick SensDof
Sensor set     Pick sens set     P-Model set     Select in feature     Select in feature	Pick SensDof
-Pick sens set -Model set -Feature -Select in feature -Select in feature	Pick SensDof
-Model set -Feature Select in feature	
-Feature gr	
Select in feature	oupall 🗸
Select in reactive	Pick a feature
DOF list	
Pick in list	Pick DOF
-Relative A-B	
-Pick Dof A	Pick +DOF
Pick Dof B	Pick -DOF
-Measure Type dis	<del>ب</del> م
-Hamonics	all
ОК	Cancel

Figure 3.3: Observation selection/generation dialog

One then gets the possibility to define an observation with the following options

- SensName A string defining the observation name that will be used as label in displays.
- **BaseJob** The job result with which the observation will be generated, used as context in the dialog following options.
- The type of observation, to be checked in the list presented below. The choice is exclusive and will expand the adequate options.
  - Sensor set To use a SensDof entry (see sdtweb sensors) that is present in the job model. Use the PickSensDof button to access the list of available entries and pick one.
  - Model set To use an integrated model based element selection. The automated selection is based on usual model accessible information. On can use a feature (as type of information) between EltSet, Mat, Pro, or groupall. Use the Pick a feature button to access respectively the EltId set list, the MatId list or the ProId list declared on the model. The groupall feature directly selects all elements in the model and thus do not need further selection.

- DOF list To use a model DOF. The Pick DOF button then provides the complete list of available DOF for the user to pick one.
- Relative A-B To generate an observation based on a relative movement between two model DOFs,  $y = q_1 q_2$ . Use the Pick +DOF to access the list of available DOF and select  $q_1$ . Use the Pick -DOF to access the list of available DOF and select  $q_2$ .
- Measure Type To provide the type of signal to generate, either disp for displacement, vel for velocity, or acc for acceleration.
- Harmonics To provide a sub-selection of harmonics retained for the post-treatment synthesis. Click on the button defaulted to all to access the list of available choices. Besides the list of harmonics present in the job result, one can choose all to retain all harmonics, odd to retain all odd harmonics only, or even to retain all even harmonics only.
- OK To validate the input and proceed to the observation generation.
- Cancel To cancel the current input and close the dialog.

# $hbm_post$

## Purpose

Commands for result post-processing. This functions provides post-treatment commands and handles the HBM result object.

# Description

## ZTraj[,Get,GetBnl,Set,SetDef]

Trajectory generation as harmonic result structures **Zcurve** from synthesis of given shapes.

• Command ZTraj generates a harmonic response associated to a given trajectory. Z=hbm\_post('ZTraj',model,def);

model is a standard SDT-NL model, def is a def curve expressed on the model DOF, typically a modeshape. The output is a HBM output Z. The model is HBM-assembled and data are initialized based on model.Stack{info,HBMOpt} or using the default options output by hbm\_solve Opt. If the def curve is real, the output will be initialized with the shape on the c1 harmonic, the pulsation is initialized by def.data(:,1). If complex, the c1 harmonic is set to the real part and s1 to the opposite of the imaginary part, the pulsation is initialized by def.data(:,1)xxx.

The output of hbm\_post ZTraj cannot directly be exploited by other ZTraj commands as an observation (see hbm\_solve AddPost) must be declared prior to generating the the HBM results object allowing response synthesis. The following command options are supported

- -res to output the result in .Res format.
- -reAss to force reassembly of the model.
- -TKT to perform model projection on def.TR.
- -harm to specify harmonics to be used for the results format (by default the first harmonic is used).
- --useq0 in conjunction with -TKT to use the static state model.Stack{curve,q0} in the trajectory, the static state will increase def.TR prior to projection.
- -qOinTR to detect the zero harmonic in def.TR (as with def.data(:,1)<1), thus initializing different generalized harmonic DOF for the zero harmonic and the others. This option is handled internally if useq0 is used.</p>
- Command ZTrajSet updates the HBM results object Z with different harmonics. XF=hbm\_post('ZTrajSet', XF, RA);

XF is the HBM results object, RA is a structure with field .harm defining the new set of harmonics.

#### hbm\_post .

• Command ZTrajSetDef only updates the HBM curve Z with a new trajectory based on the same harmonic DOF than the initial one.

XF=hbm\_post('ZTrajSetDef',XF,struct,opt,struct,d1);, with XF the HBM results object, opt the opt structure, d1 the new shape.

- Command ZTrajGet synthetizes and outputs a transient NL trajectory (in the standard fe\_timeNL format) based on one of the state of the provided HBM results object. def=hbm\_post('ZTrajGet', XF, RO);. The following command options are supported
  - nTSampval to set the number of time samples to val in the output.
  - NoT to generate a trajectory based on the model DOF.
  - iMode ind to select the state index ind in the HBM results object.
  - coefval to apply an amplitude coefficient val to the state.
  - NeelUNLval to output the non-linearity observations. Set val to a two digit number [unl vnl] set to 1 if needed, 0 otherwise to obtain def.FNL.unl and/or def.FNL.vnl.
  - initnl to perform and keep initializations necessary to optimized trajectory update by linear state coefficient application with hbm\_solve @defUpCoef.
  - NoStat to ignore unl0 and vnl0 in the observation and evaluation of non-linear forces.
- Command ZTrajGetBnl outputs the non-linear harmonic forces based on the provided HBM results object.

bnl=hbm\_post('ZTrajGetBnl',XF,RO);.

XF is a HBM results object, RO is an option structure with optional field .nTSamp to specify the time sampling number used to evaluate the harmonic non-linear forces.

#### AddPost

Handles observations declarations for post-treatments from the HBM curve output to the HBM results object.

hbm\_post('AddPost',job,data); is used to handle outputs stored in the application. job is the name on which the observation will be applied (entry in PA.Stack). data provides the observation, it can be of the following types

- A DOF list in numeric format
- An observation struct, with fields .cta an observation matrix and .DOF the associated DOF vector. struct('cta',1,'DOF',21.03,'name','tip'). Appends the SensDof entry in the model PA.Stack{job}.

• A model. If field .DOF is present, it is directly used, otherwise DOF are obtained from the .Node field.

For all cases, a **Sens** data structure is generated and a **SensDof** entry is added to the model **PA.Stack{job}**2. The GUI then adds the observation to the available observation list.

Manual handling outside the GUI requires the use of a results structure RE as a third argument. The same operations are performed, the observation list stored in RE.PostSensList being updated RE=hbm\_post('AddPost',lab,data,RE);.

#### Init

Initializes the HBM result object.

XF=hbm\_post('Init',UI,ci);.

UI is a variable pointing to the HBM solver output data. It can be left empty, in such case it is initialized by UI=hbmui('PARAMUI'), or it is a results structure with mandatory fields .Res (see hbm\_post ZTraj-Res) and .PostSensList (see hbm\_postAddPost). ci is a iiplot object that will host the results display. It can be left empty not to display results.

If results are stored in the application, the HBM result object associated to the current applcation state can be recovered using XF=hbm\_post('init').

xxx need for an integrated call Manual initialization with no display can be performed from an assembled harmonic model mo1 and a HBM curve output structure Z using

RE=struct('Res',{{Z.name,{Z,mo1}}}); RE=hbm\_post('addPost',Z.name,mo1.DOF,RE); XF=hbm\_post('init',RE,[]);

## HBM results object

The HBM results object provides methods to synthetize transfers or transient trajectories based on a HBM result, it is based on the curvemodel object that is a SDT curve wrapper. The result object is initialized using hbm\_post Init.

#### XF.[GetData,X,Y,Xlab]

Provides access to the resolved content of the HBM result object XF in its current state. X=XF.X;

**Warning:** depending on the current object state, recovering the full output can be a very intensive task and may generate a very large volume of data !

The .GetData method returns the full curve structure synthetized for the current object state. The other methods only return the field corresponding to their name.

## hbm\_post \_

## XF.set

HBM results object options handling.

XF.set('nTSamp',100);

For an app linked object, the options are available in the GUI Post tab. This programmatic way can be used in scripts interacting with the GUI, or with results objects not linked with the GUI.

## XF.Stack

Provides access to HBM results context data. The object mode can be programmatically altered through this way.

# XF.Stack'type'=val;

Accepted types for val are

- Freq To handle maximum amplitude frequency responses.
- Time To handle transient responses.
- Harm To handle Harmonic contribution responses.
- FFT-1 To handle FFT of the first harmonic transient response.

# hbm\_solve

## Purpose

HBM Solver and base utilities commands.

# Description

Variables used during the resolution are the opt structure xxx

# AFMap

Performs amplitude/frequency response MAPS, possibly in a given subspace.

# Z=hbm\_solve('AFMap',model,RO);

model is a standard SDT-HBM model. xxx By default the study is performed on the first harmonic of the model active DOF. It is possible to provide a customized harmhdof vector or even a reduced subspace by using a third argument def as a structure with fields

- .TR a structure providing a physical reduction basis, with mandatory fields .TR.DOF the full DOF vector and .TR.def the Rayleigh-Ritz vectors stored in column.
- .hdof a harmhdof vector based on the generalized DOF defined by .TR.

RO is a structure with fields xxx (that can also be provided in a preemptive way in the RO structure)

- .Freq frequency vector in Hz.
- .Amp amplitude (scaling coefficient applied to the loads). Default equal to 1.
- -bnl to store and output the NL force vector
- -iter to perform resolution iterations. By default the response is based on the response prediction induced by the non-linear forces generated by the linear response trajectory.
- -z0 to store and output the linear response.
- -itInitstra to alter the initialization strategy, set stra to either 0 to initialize at a given amplitude by the scaled linear initial state a the first amplidute, or 1 to keep the current response of the previous state.
- -fscan to perform a direct frequency scan for a given amplitude and interpolate the result on the .Freq input.
- RO.pList field can be provided, defines the parameters order in a cell array, to be used in the multiple loop from external to internal. By default set to {Amp, Freq}.
- RO.harm defines the harmonic vector to be used, by default set to 1.

• model.Stack{'info', 'HBMOpt'} Provides a custom opt structure. Beware that many fields are redefined to perform this resolution.

Output Z is a standard SDT-HBM Zcurve with multiple dimensions corresponding to Hdof, Freq and Amp.

#### Assemble[call,init,exit]

Provides and performs SDT-HBM specific pre-post assembly operations. The assembly call to be passed to fe\_case is provided by

```
st=hbm_solve('AssembleCall');
```

AssembleInit and AssembleExit are internal calls handling non-linearity initializations and proper load definitions.

fe\_time[,cleanup]

Command fe\_time is called back by fe\_timeas the base HBM solver, defined by field opt.Method in the opt structure.

Command fe\_timeCleanup performs base post-treatments to the raw solver output and handles direct storage in interaction with hbmui. By default the output will be stored in the application, and base results can directly be displayed in iiplot.

This command is usually performed as an internal command at solver exit based on the field opt.FinalCleanupFcn in the opt structure. The command then expects that the caller uses variables out, opt and model to respectively store the solver output as a HBM curve output, the solver running options (opt) and the assembled model used by the solver. An external call is also possible, using out=hbm\_solve('fe\_timeCleanup',out,opt,model).

The following command options are supported

- reset to reset iiplot before display.
- NoPlot not to disply the results in *iiplot*.
- NoUI not to store the results in the application.
- ExitFcn*cbk* to perform additional custom operations at the end of the cleanup procedure, the string *cbk* will directly be called by the **eval** function.
- -rethrow to output the result data structure.
- -cfval to provide a specific iiplot figure. If val is not strictly positive the default iiplot figure (or last active one) will be used, otherwise the figure with given positive handle will be used. xxx hbmui('PARAM.cf') xxx

## harm[lab,hdof,place,c]

Harmonic handling utilities. This series of functions provide tools for HDOF definition handling and matching.

- harmHdof defines a harmonic DOF (HDOF) vector.
   hdof=hbm\_solve('harmHdof', model, harm); model is a standard SDT model from which the active DOF will be recovered, using model.DOF if present, or fe\_casegetTDof. It is possible to provide a DOF vector instead of a model. harm provides the harmonics numbers to be used, set to 1 if omitted. To select only specific time functions, it is possible to use a regular expression token as a third argument.
- harmC performs HDOF localization utilities.
   c=hbm\_solve('harmC',hdof,sdof) provides an observation matrix c to observe harmonic DOf subset sdof in hdof.
   sdof=hbm\_solve('harmC',hdof,sdof,typ,in) respectively provides

  - typ='dof', in=1 the intersection of hdof and sdof .
  - typ='ind', in=1 the indices in hdof interecting with sdof.
  - typ='dof', in=2 the harmonic DOF of hdof not present in sdof.
  - typ='ind', in=2 the indices in hdof no present in sdof.

Command option hID performs the match on the time function label only.

- harmPlace Places a harmonic def (defined on a set of harmonic DOF) on the set of harmonic DOF hdof.
   d1=hbm\_solve('harmPlace',hdof,def);.
- harmLab A rather internal subfunction generating time dimension labels. lab=hbm\_solve('harmLab',1:3).

```
model=demosdt('demoUBeam');
hdof=hbm_solve('harmHdof',model,0:3); % adof, {c0,c1,s1,c2,s2,c3,s3}
hdof1=hbm_solve('harmHdof',model,1,'s'); % adof, s1
```

```
i1=hbm_solve('harmC',hdof,hdof1,'ind',1);
sdof=hbm_solve('harmChID',hdof,'c1','dof',1);
isequal(hdof1,sdof)
```

#### hbm\_solve \_

#### Opt

Solver options handling. Reference to data fields is provided in opt for initialization and opt for values used during the solve.

```
opt=hbm_solve('opt');
opt=hbm_solve('opt fstart=.1 fend=100',opt);
RA=struct('fstart',.5,'fend',100,'dsmin',.001);
opt=hbm_solve('opt',[],RA);
opt=hbm_solve('opt',opt,RA);
```

#### Reduce

Performs model reduction with proper handling of non-linearities.

[model,Case,Load]=hbm\_solve('Reduce',model,TR);

model is a SDT-HBM model, TR is a def structure defining the reduction basis.

The model will be fully assembled then reduced, it is possible to provide a pre-assembled model. To do so, it is necessary to define the associated case by providing it in a third argument. The output is an assembled reduced model complying with initialization of SDT-HBM resolution procedures.

#### @abscHBM

Performs a state prediction based on an interpolated curvilinear frequency step.

[Z,opt]=abscHBM(Z0,opt,j1);

ZO is the initial state provided to the solver, opt is the internal solver structure, j1 is an interation indicator.

The outputs are Z a predicted state, and opt with updated curvilinear frequency step opt.dS.

For the first iteration j1==1, the initial state Z0 is output.

For the fixed strategy opt.dsOpt.step==0, the curvilinear frequency step is set to 0.5\*opt.dsOpt.dsfix, and the predicited state is the initial state Z0 with the new frequency.

For the Lagrange interpolation strategy opt.dsOpt.step==1, the curvilinear frequency step is defined as being between opt.dsOpt.dsmin and opt.dsOpt.dsmax and updated to optimize the number of iterations towards the target opt.dsOpt.iopt. The update is performed by applying a multiplicative coefficient to the current frequency step opt.dS as the ratio between the optimal iteration number opt.dsOpt.iopt and the last number of iterations opt.ite. The multiplicative coefficient is bounded between opt.dsOpt.bmin and opt.dsOpt.bmax. The predicted state is the result of the Lagrange interpolation of degree opt.dsOpt.Ldeg of the last states at the incremented frequency.

#### **@ATimesZ**

Computes the linear dynamic harmonic forces o1=A\*z0 in an implicit manner (*i.e.* without actually building the dynamic harmonic stiffness matrix).

o1=ATimesZ(model,Case,opt,z0)

The output o1 is the linear dynamic harmonic forces.

## @buildA

Initialization and/or generation of the harmonic dynamic stiffness.

opt=buildA(model,Case,opt,Z,i1).

model is a SDT-HBM assembled model, Case is the associated case, opt is the internal solver running option structure, Z is the current harmonic state; these variables must have been initialized by hbm\_solve InitHBM. i1 is the output option:

- i1=0 will output opt with field opt.A initialized as the dynamic harmonic stiffness matrix.
- i1=1 will output opt=A as the dynamic harmonic stiffness matrix.
- i1=2 performs optimized initialization of the dynamic stiffness matrix as a cell array of component matrices, so that the frequency dependency can be represented as a weighted sum of the component matrices. If opt.A is empty an initialization at unit frequency is performed, if opt.A is a cell array the actual dynamic stiffness matrix is directly output (opt=A).

The frequency is recovered from context, either opt.Opt.fvar==1 and the pulsation is taken as the last value of the harmonic state (w=Z(end)), or opt.Opt.fvar==0 and the pulsation is directly taken as w=opt.w.

The output opt is either the running option structure with filled opt. A or directly the harmonic stiffness matrix.

## @buildCHarm

Generation of interlaced harmonic observation or command matrices.

[c,harm]=buildCHarm(c,opt,typ).

c is an observation or command matrix, opt is the internal solver running options that must have been initialized by hbm\_solve InitHBM, typ provides the type of matrix c. Either typ='c' to declare an observation matrix, or typ='b' to declare a command matrix.

The output is the interlaced observation or command matrix c. Line ordering is interlaced (*i.e.*) the sequence is first the harmonics, then the initial line order. harm is the retained harmonics. For command matrices, Fourier coefficients are applied to the matrix terms.

It is possible to define a field **opt.subH** with a sub-set of harmonics to be used for the observation generation, independently from the initial **opt.harm** field that is used by default.

#### hbm\_solve \_

#### @buildHkt

Generation of time function space.

# opt=buildHkt(opt);

opt is the internal solver running option initialized by hbm\_solve InitHBM.

The output is the internal solver running option opt with updated fields opt.Hkt, opt.dHkt and opt.Htk = opt.Hkt', that are the time harmonic components.

#### @buildLoad

Prepares the external load structure for the HBM solver, based on the usual transient Load representation of fe\_time.

## Load=buildLoad(Load);

The output has a resolved field .adof providing the harmonic DOF used to described the external load, a command matrix .b, a field .curve providing the optional variations of external loads with the frequency.

Load is a load structure complying to time simulations. See fe\_load, fe\_load buildu.

#### @ctaSubH

Integrated generation of a subharmonic observation matrix based on a HBM output curve. opt=ctaSubH(cta,d0,subH);

cta is an observation matrix, d0 is an HBM output data structure, subH is a sub-harmonic selection. subH is either directly a vector of harmonics, or a string token set to

- all to retain all harmonics of d0.harm.
- odd to retain only the odd harmonics of d0.harm.
- even to retain only the even harmonics of d0.harm.

The output is the opt data structure with added or updated fields .cta, opt.iadof, opt.idof, opt.harm, opt.hVect, opt.hId to comply with the subharmonic selection.

## @defUpCoef

Optimized trajectory linear amplitude change.

# d2=defUpCoef(d2,Amp);

d2 is a trajectory initialized the hbm\_post ZTrajGet-initnl, Amp is a scalar amplitude coefficient. The output is d2 the trajectory linearly set to amplitude Amp, relative to d2.coef, the initial amplitude.

The trajectory and non-linearity observations are updated using Amp/d2.coef, non-linear forces are re-evaluted to provide a linearly updated trajectory.

## InitHBM

The subfunction @intHBM performs HBM solver specific initializations based on a SDT model with non-linearities. [model,Case,opt,ZO,Zf]=initHBM(model,Case,opt,u,fext); model is a standard fully assembled SDT-NL model, Case is the corresponding resolved Case structure, opt is the solver running option structure opt, u is a system state expressed on Case.DOF, usually a static state, fext is the external Load structure expressed on Case.DOF. model, Case and fext should be compliant to the SDT-HBM data structures, and can be typically obtained from usual models with the provided hbm\_solve AssembleCall: [model,Case,fext]=fe\_case(hbm The outputs are data ready to be used in HBM solvers, opt is completed, ZO is the initial state, Zf is the current harmonic load vector.

The initialization procedure

- prepares the internal solver running parameters,
- prepares system matrices and initializes the harmonic dynamic stiffness structures,
- initializes the initial harmonic state and state buffers for prediction and interpolation,
- builds external forces harmonic vectors,
- builds the harmonic interlaced observation and command matrices of each non-linearity.

## @iterHBM

Performs an interative resolution for a given initial state.

## [Z,ki,opt]=iterHBM(ki,Zf,Z,model,Case,opt,j1);

ki is for the moment reset internally and can be provided empty. Zf is the harmonic load vector, Z is the initial harmonic state, model is the assembled model, Case the correspoding case structure, opt the solver running options. All these variables must have been initialized with hbm\_solveInitHBM. j1 is a step indicator, also used in the base solver to know whether the state buffer has been fully filled.

The outputs are Z the resolved HBM state,  $\mathtt{ki}$  the current Jacobian, and  $\mathtt{opt}$  the internal solver option structure.

## @getZf

Generates the harmonic external forces vector from the Load context. This is used by hbm\_solve @resHBM.

#### Zf=getZf(Zf,opt);.

Zf is the current load data structure, or a resolved harmonic vector. Nothing is performed in the latter case. opt is the internal solver running option. In particular the current pulsation is found in opt.w.

#### hbm\_solve \_

#### @outputInitHBM

Initializes the solver output structure, to be filled by hbm\_solve outputHBMFcn.

[out,opt]=outputInitHBM(model,Case,opt);

model is unused at the moment, Case is the case corresponding to the harmonic model, opt is the internal solver running option. These variables must have been initialized by hbm\_solve InitHBM.

#### @outputHBMFcn

Output data structure (pointer addressed) filling.

# outputHBMFcn(out,Z,opt);

out is an output data structure initialized by hbm\_solve @outputInitHBM, Z is the current state to be possibly stored, opt is the internal running option solver.

There is not output associated to this command as input structure out fields are handled by pointer. The current state is stored if its associated frequency has changed in absolute value by more than opt.SaveFreq.fStep from the last saved frequency point (out.X2(out.cur(2))). The frequency is recovered from context, either opt.Opt.fvar==1 and the pulsation is taken as the last value of the harmonic state (w=Z(end)), or opt.Opt.fvar==0 and the pulsation is directly taken as w=opt.j1. In this case, opt.j1 is differenciated from opt.w to allow cutomized handling of the saving strategy.

#### @resHBM

Computation of the harmonic balance residue and associated finite differences Jacobian.

[r,ki]=resHBM(Zf,Z,model,Case,opt,ki,jite);

Zf is the harmonic load vector, Z is the current harmonic state, model is the harmonic model, Case is the corresponding case, opt is the internal solver running option, ki is the current Jacobian, jite is a current iteration indicator.

The outputs are  $\mathbf{r}$  the harmonic residue, and  $\mathbf{k}\mathbf{i}$  the associated Jacobian.

The Jacobian is computed by finite differences using a dZ amplitude defined by opt.epsi and initialized at 1e-9. It is computed if ki is empty, or if opt.JacobianUpdate is not null.

It is possible to obtain the external harmonic forces of the current model by setting Z to empty. In such case, the non-linear forces will be obtained from model.FNL and summed with -Zf in the output r.

# hbm\_utils \_\_\_\_\_

## Purpose

Utilities used for HBM development and administration.

## Syntax

```
hbm_utils CommandString
hbm_utils('CommandString')
```

## Description

hbm\_utils deals with the paths handling (Path command), the generation of the documentation (Latex command) and other utilies.

## Path

hbm\_utils('Path') fixes MATLAB path and sdtweb('\_path') to include DYNAVOIE based on the result of which('hbm\_utils'). The expected directory structure is detailed in section 2.1. Note that you should

## Help

You can automatically update, your documentation files using hbm\_utils('HelpGet') which will get a zip file from the server and decompress it into DynRootDir/help. The help contains both the PDF and HTML files which can be opened with the sdtweb function, for example:

sdtweb('hbm\_utils') % requires sdtweb >1.50

Note that you possibly have to configure your proxy adress in your Matlab preferences (File/Preferences/W

#### Latex, Hevea

The following commands are used by SDTools for the maintenance

- Latex compiles the documentation using pdflatex.
- pdf opens the manual in a browser window.
- hbm\_util('HelpPut') updates the zip file containing the documentation

## Verbose\_Mode

Verbose mode mecanism. Now one can different levels of warning:

```
hbm_utils('Verbose_Mode',-1) % SILENT
hbm_utils('Verbose_Mode',0) % NORMAL
hbm_utils('Verbose_Mode',1) % VERBOSE
```

## $hbm_utils$ \_\_\_\_\_

# Distrib

**DistribGen** is used by SDTools to generate a protected copy of DYNAVOIE, which can be sent as a zip file. **DistribPatch** is used by SDTools to generate a SDT patch relative to the last reference version of SDT.

# nl\_spring

# Purpose

Non linear links/force modeling for time simulation

# Syntax

```
model=nl_spring('tab',model);
...= nl_spring('command', ...)
Opt=nl_spring('TimeOpt');
```

# Description

nl\_spring supports non-linear connections and loads for transient analysis. Non linear springs between 2 DOF (see nlspring). loads which depend on DOF values (see DofKuva, DofV), springs between 2 nodes in different bases (see RotCenter), etc. ...). A full list of non-linearities is given in nllist

Standard non-linear simulations are handled by **nl\_solve**. Below is a description of the inner mechanisms of a non-linear simulation with the non-linear toolbox.

After the non linearity definition, a proper TimeOpt is required to set the good fe\_time calls to perform a non linear Newmark time integration. A default TimeOpt can be set using nl\_spring TimeOpt. It is possible to save transient results on the fly using a properFinalCleanup call, see nl\_spring fe\_timeCleanupCall, and to reload the same results using fe\_simul fe\_timeLoad. The following steps are required for a time simulation

• Definition of non-linear properties. These are stored as pro entries of the model stack. The associated property function must handle non-linearities which is currently only the case for p\_spring and p\_contact.

A non-linearity is always associated with elements or superelements (typically a celas element. A given group of elements can only be associated with a single non-linearity type.

The information needed to describe the non linearity is stored in a .NLdata field.

- Model initialization using the an fe\_case('assemble') call in fe\_time, is followed by the building of a model.NL stack that describes all non-linearities of the model in a format that is suitable for efficient time domain integration. This translation is performed by the nl\_spring NL command.
- Jacobian computation, see nl\_spring NLJacobianUpdate.
- Residual computations are performed through mkl\_utils. The nominal residual call is r=-fc; mkl\_utils('residual', r,model,u,v,a,opt,Case);.

# Supported non linearities

See nllist for supported non linearities, and nl\_fun to add your own non-linearities.

#### ConnectionBuild

One can define a set of non linear links between 2 parts of a model using a call of the form [model,idof]=nl\_spring('ConnectionBuild',model,data);

idof is a second optional out argument. It returns the list of DOF concerned by links (it can be useful in order to reduce super elements keeping idof as interfaces DOF for instance). data contains all the information needed to define links. It is a 3 column stack like cell array. First column contains the string 'connection', the second the name of the non linear link described in the third column that contains a data structure with following fields:

- .Ci define nodes to connect in first (.C1) and second component (.C2). It can be a vector of NodeId or a screw data structure (slave nodes of the model nodes via RBE3 links, see see sdtweb('fe\_case#connectionscrew').
- .link defines how to link component 1 to component 2. It is a 1x2 cell array. First cell defines the type of link ('EqualDof' or 'Celas') and the second gives information about the link. For celas link it is a standard element matrix row with 0 replacing NodeId :[0 0 DofId1 DofId2 ProId EltId Kv Mv Cv Bv].
- .NLdata (optional) defines non linearity associated to celas link. See the list in list of supported NL. If this field is not present or empty, only linear link is considered.
- .PID (optional) is a 1x2 line vector that defines PID (second column of .Node matrix, see sdtweb('node') of connected node (1rst column for 1rst component).
- .DID (optional) is the same as above, defining DID (third column of .Node matrix, see sdtweb('node') of connected nodes.

Following example defines a model with a cylinder and a hole in a block. The cylinder is linked to the block by 3 celas preserving the pivot link.

```
mo1=demosdt('demoConnection-vol'); % meshes models
mo1=fe_case(mo1,'fixdof','base','z==-1'); % clamps the cylinder base
r1=struct('Origin',[0.5 0.5 0.5],'axis',[0 0 1],...
'radius',.1,'rtol',.01,'length',1,'Npt',-3,...
'ProId',111,'planes',[]); % Cylinder-side
r1=nl_spring('ConnectionCyl',r1); % defines planes
r3=r1; r3.ProId=1; % Block-side
link={'connection','link1',struct('C1',r3,'C2',r1,...
'link',{{'celas',[0 0 12345 12345 1000 0 1e9]}})}; % Defines connection
[model,idof]=nl_spring('ConnectionBuild',mo1,link); % builds connection
cf=feplot(model); % displays in feplot
fecom promodelviewon; fecom('curtab Cases','link1_2');
```

```
def=fe_eig(model,[5 20 1e3]); % computes the first 20 modes
if length(find(def.data<1e-3))>1; sdtw('_err','connection failed'); end
cf.def=def; fecom ColorDataAll % displays modes
```

See also t\_nlspring('2beam') example.

# ConnectionCyl

Utility to fill the .planes field of a cylinder connection in the standard connection screw data structure format (see fe\_caseg ConnectionScrew). dataOut=nl\_spring('ConnectionCyl', dataIn);

The dataIn uses fields:

- .Origin origin of the cylinder axis, .axis orientation of the cylinder
- .rtol radius tolerance for cylinder selection.
- .length length of the cylinder.
- .Npt number of planes (equally distributed on the whole length). If Npt<0, ends of the cylinder are included in the connection points.
- .ProId ProId of the elements containing nodes to connect.



Figure 3.4: ConnectionCyl

## nl\_spring

InitV

## q0=nl\_spring('InitV',model,d0,R0);

InitV computes the initial static displacement and velocity associated to a DOF initial position and velocity. d0 is a data structure with field .DOF containing the DofId where initial value is applied and .def containing initial displacement and velocity at this DOF. RO is a optional input argument data structure with following fields that define:

- .dt time step for time integration.
- .dq increment for initial vel computation.
- .Nv] number of time steps to reach d0.def(1) (displacement is imposed as a 0.5(1 cos) time function on these time steps).
- .Np number of steps to stabilize at d0.def(1) and d0.def(1)+dq.

If input argument RO omitted, options are get from 'info' 'initvopt' Stack entry. If there is no such entry, InitV parameters are computed using -optim process (see below). Displacement at q0 and q0+dq is obtained meaning the last Np/10 steps of each stabilization period, and initial velocity is computed from those 2 displacements to match d0.def(2) at d0.DOF.

[q0,R0]=nl\_spring('InitV-optim',model,d0); can be used to find input parameters R0. Optimization of dt and Np is performed from given or default values. Parameters dq and Nv are kept at given or default value. First dt is optimized. dt is increased (multiplied by 4) until time integration of the InitV process diverge and last dt that leads to convergence is kept. Then Np is increased by 100 steps until the deformation is converged on the stabilization periods, that is to say that a criteria taking in account standard deviation/mean of the deformation and the ratio of the last Np/10 steps upon previous Np/10 steps on each Np period is less than a tolerance (2.0).

See 2beam example XXX.

#### NL

## model=nl\_spring('NL',model)

This command is used to build .NL field data for time integration from NLdata field in NL p\_spring property entries in the input model Stack. The command option -storefnl can be used to specify the way of computing and storing a non linear effort associated to NL (for those which support it).

## NLJacobianUpdate

opt.Jacobian=nl\_spring('JacobianCall') returns the callback used to update or initialize the Jacobian ki used in iterative methods. The said jacobian must take non-linearities into account.

For the case of a non-linear spring, the most important gradient of the tabulated law Fu is added as stiffness between the 2 DOF to the stiffness matrix and the most important gradient of Fv to the damping matrix.

For non-linear iterations in a Newmark scheme, the Jacobian is given by

ki=(model.K{3}+kj)+ (opt(2)/opt(1))/dt\*(model.K{2}+cj) + 1/opt(1)/dt^2\*model.K{1};

Accepted command options, associated to variants of the call are

- There are three outputs accessible, being [ki,mo1,C1]=nl\_spring('NLJacobian'...).
- -noFact not to factorize the output Jacobian. This is useful if further actions are performed on the Jacobian after the standard call.
- -TangentMdl to return tangent model. It is assumed that model.K(1:3) correspond to M, C, and K (in this order). u and v variables of caller workspace can be needed.
- -TangentMdl-back to return a superelement containing the tangent matrices.
- -TangentMdl-back-sepKj to return a superelement containing the tangent matrices split by non linearities.
- -ener to compute for each def stored in model.d1 def structure (that is typically computed modes), some associated energies:
  - frequency in Hz.
  - damping damping ratio:  $(\phi_j^T[C]\phi_j)/(2\omega_j)$ .
  - enerK total strain energy:  $\phi_j^T[K]\phi_j$ .
  - enerC  $\phi_j^T[K]\phi_j$ .
  - *NLlink*-enerK strain energy for each NL link:  $\phi_i^T[K_{NLlink}]\phi_j$ .
  - *NLlink*-enerK for each NL link:  $\phi_j^T[C_{NLlink}]\phi_j$ .

## SetPro

## model=nl\_spring('SetPro ProId i ParamName1 Value1 ...',model)

This command is used to change some nl\_spring properties parameters. i is the ProId of corresponding p\_spring property, *ParamName* the name of parameter to change (k for il(3), c for il(5) or the field name in NLdata) and *Value* the value to assign.

#### nl\_spring

It is possible to define a new property by specifying an NLdata structure in third argument: model=nl\_sprin ProId *i*', model, NLdata). If the property already exists, the NLdata is interpreted as a string of parameters and parsed to define the fields specified in the given NLdata to the existing one. Command option Edit allows directly merging the existing NLdata to the provided NLdata with priority given to the new fields.

```
model=nl_spring('Demo1DOF');
% define a non linearity with partial definition of parameters and other by default
NLdata=nl_fun('db data 4') % standard NLdata defintion
% NLdata has fields data, Jacobian (by default) and type
% set in model
model=nl_spring('setpro proid201',model,NLdata);
% edit the nl_fun nl by string keyword
model=nl_spring('setpro proid201 data2',model);
% edit the nl_fun with struct input
% property will be parsed using nl_fun('paramedit')
model=nl_spring('setpro proid201',model,struct('Jacobian',2));
% field Jacobian has been edited, other fields are kept unchanged
model.Stack{end,3}.NLdata
model=nl_spring('setpro proid201',model,struct('NewField','test'));
% you can see that in this case NewField was not set
% as it is not referenced in the nl_fun parameters
model.Stack{end,3}.NLdata
% Force the with struct input with no check
model=nl_spring('setproedit proid201',model,struct('data',10,'NewField','test'));
% in this mode the NewField is propagated regardless of the
% standard nl_fun input
model.Stack{end,3}.NLdata
```

Standard NLdata structures depend on the non-linear function, see nllist for more details. They can be obtained through the nl\_function command db, see nl\_fun for more details. In the case where

#### GetPro

#### pro=nl\_spring('GetPro',model)

This command is used to get non linear properties in the model stack.

- Command option ID allows getting a specific non linear property by specifying its ProId.
- Command option type '`nl\_fun'' allows getting the non linear properties of a specific type. See nllist for more details on types of non-linearities.

## Follow

The Follow mechanism can be used to observe some variable evolution during the time integration. opt=nl\_spring('Followi', opt);

1st Follow consists in monitoring the number of iteration, the residual norm and displacement increment norm at each time step.

```
model=nl_spring('Demo1DOF')
opt=stack_get(model,'info','TimeOpt','GetData');
opt=nl_solve('TimeFollow1',opt); % niter norm(r) norm(dq)
def=fe_time(opt,model);
```

2nd Follow consists in monitoring the def.FNL in iiplot. For the moment the mechanism is different (so note that you can't both tracker niter and FNL), and you only have to specify the field .Fnlliplot equal to 1 in the 'info', 'OutputOptions' stack entry of the input model, as in following example :

```
model=nl_spring('Demo1DOF');
r1=stack_get(model,'info','OutputOptions','GetData');
r1.Fnlliplot=1; % define FNL tracker
model=stack_set(model,'info','OutputOptions',r1);
opt=stack_get(model,'info','TimeOpt','GetData');
def=fe_time(opt,model);
```

See sdtweb t\_nlspring('Follow') for list of wanted cases.

## TimeOpt

This command returns usual default TimeOpt for non-linear simulations. By default the output is the same as the TimeOptNLNewmark presented below. See also fe\_time for TimeOpt definition details. Supported TimeOpt commands are

- TimeOptNLNewmark, or TimeOpt to obtain the TimeOpt for NLNewmark simulations. Use TimeOpt-gam .51 to introduce numerical damping by directly giving gamma.
- TimeOptStat to perform static simulations (see also fe\_time nl\_solve).

## nl\_spring

- TimeOptTheta to perform time simulations with the  $\theta$ -method (see fe\_time). Numerical damping can be introduced using TimeOptTheta-alpha .05, the specified  $\alpha$  value will be added to  $\theta$ , so that the coefficient used in the simulations will be  $\theta_1 = \theta + \alpha$ .
- TimeOptExplicit to perform time simulations with the explicit Newmark scheme.

The following command options allows setting other TimeOpt fields to their desired value.

- dtval time step.
- tsN number of time steps.
- tendval optional end time
- tInitval initial time.
- AlphaRval a global Rayleigh damping mass coefficient (applied to the model total mass).
- BetaRval a global Rayleigh damping stiffness coefficient (applied to the model total stiffness).
- maxNoutN requests an output subsampling strategy such that only N times equally spread over the simulation time span are output.
- RelTolval requests a specific relative tolerance for the convergence of iterative schemes.
- -gammaval requests a specific  $\gamma$  coefficient (default to .5) of the Newmark scheme. For the non explicit versions,  $\beta$  is adapted to ensure unconditional stability of the scheme.
- -thetaval requests a specific  $\theta$  coefficient (default to .5) of the Theta method.
- -acallstr provides a series of command options applied to the AssembleCall generation.
- -fcleanstr provides a series of command options applied to the FinalCleanupFcn generation.
- -jcallmodel edits the jacobian call to allow late model modification.

Alternatively to providing all these command options in the command string, one can provide a MATLAB struct with equivalent fields as an additional argument.

By adding an SDT model as third argument, the generated **TimeOpt** will be directly integrated in the model, that will be output.

Sample calls :

```
% basic call
opt = nl_spring('TimeOpt dt1e-6 ts3e5 maxNout1e4 -acall"lumpedMass"');
% call with struct input
RO=struct('dt',1e-6,'ts',3e5,'maxNout',1e4,...
'acall','lumpedMass');
opt = nl_spring('TimeOptExplicit',RO);
% basic call with model input
model = nl_spring('TimeOpt dt1e-6 ts3e5 maxNout1e4 -acall"lumpedMass"',[],model);
% call with struct and model input
model = nl_spring('TimeOptExplicit',RO,model);
```

Convergence tests depend on the iteration algorithm and several behaviors can be obtained by modifying RelTol. In any case the absolute value of RelTol is used for the convergence test application; its sign is used to determine the convergence test to be used as described in the following.

- For algorithms using iterNewton as IterFcn, as is the case for methods newmark (explicit or not), NLNewmark, and staticNewton.
  - using RelTol > 0 tests the convergence of the mechanical residue, relative to value opt.nf. If opt.nf is not provided, the scheme takes in input the norm of the external forces fc at the first time step, or if zero the norm of the first residue of the first time step. If still zero, opt.nf is set to 1. This convergence test is the most widespread as it ensures mechanical stabilization. It is strongly recommended for static computations, or when using large time steps.
  - using RelTol < 0 tests the convergence of the displacement correction, relative to the current displacement norm. The idea of this mode is to stop iterating if the correction becomes negligible, this is very useful to limit iterations with little impact on the results in transient simulations with small enough time steps. This must be used with care as this criterion does not imply that the mechanical residue is converged at the end of the time step, it is thus strongly advised to check results convergence.</p>

iterNewton does not support the use of opt.cvg yet.

- For algorithms using itertheta\_nl as IterFcn, as is the case for method theta,
  - using RelTol > 0 tests the convergence of the velocity field, its correction relative to the previous iteration velocity norm.
  - using RelTol < 0 tests the convergence of the velocity field, and the model.FNL vector, their correction relative to the previous iteration norm. Stabilization of the model.FNL field may be difficult to attain and very sensitive as this vector can contain heterogeneous data, this mode is then not recommended by default, and use of opt.cvg should be preferred.

iterthetal\_nl supports the use of opt.cvg, that forces iteration if set to 1. It is reset to zero at the start of each iteration, but any non-linearity can alter its value by using sp\_util('setinput',opt.cvg,ones(1),zeros(1));. Each non-linearity can thus internally test the convergence of its fields of interest and apply a convergence veto if its convergence is not satisfied.

From standard fe\_time simulations, the following TimeOpt fields are added or modified

- Jacobian field is modified to take into account non linearities, see NLJacobianUpdate.
- Residual field is modified to take into account non linearities, and to use mkl\_utils to improve computation times, see sdtweb mkl\_utils. This should be initialized by nl\_spring('ResidualCall'
- AssembleCall field is modified, to perform non-linearities initialization after assembly. AssembleCal is the string passed to fe\_case, generated by nl\_spring('AssembleCall').
- OutputInit field is modified to also check non linearities and initialize non-linearities related outputs, this is a callback generated by nl\_spring('OutputInitCall').
- FinalCleanUpFcn field is modified to perform cleanup on non linearities as well, this is realized through the ExitFcn command option of fe\_simulfe\_timeCleanUp (see fe\_timeTimeOpt), using '-ExitFcn"nl\_spring(''fe\_timeCleanUp'')"'. This should be initialized by nl\_spring('fe\_t
- OutputFcn The output function should be generated by the OutputInit command, since it handles proper interpolation of output as function of the time step, and requires fine tuning in the case of non linear simulations. If nl\_spring handles the OutputInit call, OutputFcn is thus reset during initializations. Handling of output time steps using a time vector in OutputFcn is supported.

#### AssembleCall

The TimeOptAssembleCall must use the -InitFcn callback of fe\_caseg Assemble to perform initialization of the non linearities.

Command options are available to tweak the assemble call with minimal user input

- MVR To adapt the assemble call for preassembled reduced models. This typically removes the -load command option of the call as this has to be recovered in the MVR itself.
- skipMKL No to transform the model matrices into mkls objects.
- lumpedMass To adapt the mass matrix mattype to 20 and get a lumped mass matrix.
- compose For more complex calls one can redefine from scratch the assemble call line to which the ad hoc initFcn will be added.

## ResidualCall

The **TimeOptResidual** callback should be a call to **mkl\_utils**, that performs optimized matrix vector products, and the computation of non linear forces handled by **nl\_functions**. Command options allows choosing a call adapted to the type of simulations

- by default a call adapted to the nlnewmark scheme.
- ResidualCallStatic provides a residual adapted to the newton-Raphson schem.
- ResidualCallExplicit provides a residual adapted to the newmark explicit scheme.

#### fe\_timeCleanupCall

The TimeOptFinalCleanupFcn callback must use the -ExitFcn of fe\_simulto perform post treatments of non linearities. Custom options classical to the fe\_simulFinalCleanup call can be added either in the command string or as a string in second argument.

```
opt.FinalCleanupFcn=nl_spring('fe_timeCleanupCall -cf-1-fullDOF');
% equivalent call with second argument
opt.FinalCleanupFcn=nl_spring('fe_timeCleanupCall','-cf-1-fullDOF');
```

In addition to the standard fe\_simulFinalCleanup, the following command options are available (to be specified outside the ExitFcn callback.

- -HDFSave To save the output in a temporary file, and output a v\_handle pointer to the saved data. This is useful for RAM optimization matters.
- -HDFfname fname In combination to -HDFSave, to specify the file in which the output will be saved.
- -Save To save the output in a temporary file, but keep the results.
- -fname fname In combination to -Save, to specify the file in which the output will be saved.

## OutputInitCall

The OutputInit callback is locked for internal nl\_spring use. Several command options are available that will be forwarded to the OutputInit procedure

- -BlockSaveN To initialize a bufferization of the output of size N. Results will be saved as blocks containing each N saved time steps.
- -exit To force exit after initialization. This can be used to check the output format without performing the simulation.
- -postFcn To provide a callback that can tweak the output at the end of the OutputInit procedure. This can be used for example to initialize out.Post post treatments.

# nl\_spring

#### TimeOutputOptions

Fine tuning of fe\_time output can be achieved by specifying an 'info', 'OutputOptions' case entry.

Accepted fields for the OutputOptions structure are

- .FnlAllT if defined and equal to 1, non-linear loads are saved at all time steps.
- .Fnlliplot if defined and equal to 1, non linear loads are displayed in an iiplot figure as curve FNL. If the display timer associated with this figure does not stop automatically, you can stop it with cingui('TimerStop').

#### mkl\_utils

Non linearities are treated by mkl\_utils mex file. Details are provided in mkl\_utils.

#### rheo2NL

OBSOLETE. Use now nl\_spring NL.

NL=nl\_spring('rheo2NL',model,DOF,offset);

This command is used to convert rheological data into a structure of data understandable for NLforce command. DOF is the list of the DOF coherent with u and v arguments of NLforce command. Offset is optional. It is a structure of data with fields .DOF and .def that defined 0 reference for Fu and Fv tab laws.

#### tab

model=nl\_spring('tab',model);

This command is used to convert formal rheological description data stored in model.Stack to a tabulated law description. The format is likely to change due to optimization of the compiled functionality in mkl\_utils (see mkl\_utils).

#### BlockSave,BlockLoad

Undocumented intermediate save of a time block for long simulations that do not fit in memory.

# $mkl_utils$

# Purpose

For detailed callback information see sdtweb('nlspring\_timeopt').

# Residual

Residual command is used to compute standard residue.

mkl\_utils('residual',r,model,u,v,a,opt,Case); call modifies variable r in memory according to following standard residue computation (implicit Newmark).

```
r = model.K{1}*a + model.K{2}*v + model.K{3}*u - fnl -fc;
```

Typically in fe\_time computations one has

opt.Residual='r=-full(fc);mkl\_utils(''residual'',r,model,u,v,a,opt,Case);';
with fc the time load (resulting from DofLoad entries in model Case) and fnl is the sum of the
non linear efforts (if any) computed directly by mkl\_utils (rotcenter, mocirc2), in the non linear
functions (see sdtweb nl\_fun) or in nl\_spring. mkl\_utils then calls the adequate nl\_fun function
(nl\_spring by default) automatically.

Such call stored in **opt.Residual** is filled by **nl\_spring('TimeOpt')** for default simulations. Model information specifically supported by the residual command are

- opt.Rayleigh if the field exists defines a global Rayleigh damping and opt.Rayleigh(1)\*model.K{1 is added to the residual.
- $model.K{2}$  can be a data structure describing modal damping with following fields:
  - .def :  $M\Phi$  vectors as columns.
  - .data :  $c_j$  modal damping coefficients as a vector.  $c_j = 2\omega_j\zeta_j$ . A second column has to be set to zero for transient applications.
  - .type : @nl\_modaldmp handling function for callbacks. The following callbacks must be handled
    - \* matrix projection  $tkt = T^T KT$ : tkt=feval(K.type,'getTKT',K,T,Tt,typ) with K the implicit matrix, T the right projection matrix, Tt the left projection matrix (can be empty or skipped if  $Tt = T^T$ , typ the output type, either *imp* to keep the implicit format (by default), or *full* to recover a full numeric matrix (to be reserved for small output sizes).
    - \* vector application f = Kq : f=feval(K.type,'getForce',K,q)
      with K the implicit matrix, q a deformation vector.

- .UseDiag : to be set to one if one wants the output of getTKT to be diagonal (as for a standard dtkt call).
- .K : optional additional damping matrix. This matrix must be in a mkl transposed
   v\_handle format (use v\_handle('mklst',K) to convert a matlab matrix to this format).
   Note that model.K{2}.K is taken in account for the Jacobian computation whereas modal damping is not.
- .defT : the resitution matrix (left side  $M\Phi$ ), that can occur mainly in the case where a non-symmetric projection has been carried out. *E.g.*, the implicit representation of  $T_l^T M\Phi \left[ \langle 2\zeta_j \omega_j \rangle \right] \Phi^T M T_R$  will use field .def to store  $T_R^T M\Phi$  and .defT to store  $T_l M\Phi$ .

Corresponding additional residue term is  $\sum_{j} [M] \phi_{j} * c_{j} * \phi_{j}^{T} [M]^{T} * v.$ 

• model.NL can be a stack of non linearities. Column 3 provides a structure with the following standard fields, see nldata.

Typically, fnl is computed by non linearity functions, see nl\_fun for details on these functions. The non linear functions are called by mkl\_utils to provide the value of fnl at a given state. Two implementations are supported

- An optimized *input-output* formulation, using observation and command matrices c and b documented in nldata. The computation of the observation is possible either on the displacement, the velocity or both, and the command is added to the residual using r = r + b\*unl. With unl a vector depending on the observation (c\*u, c\_v v).
- A used defined addition (older format, that should be only used when the generic *b,c* format fails to be relevant. In this mode the non linear function must add fnl by itself, choosing the sign convention, using a call of type of\_time(-1,fc,fc-fnl);. One will note that the residue vector is named fc in the non linear functions.

## Install

The mkl\_utils function is distributed using xxx In order to make proper distribution of mkl\_utils please send us result of:

```
ver
cd(fullfile(matlabroot,'bin',getenv('MATLAB_ARCH')))
!ls -lt *mkl*
```

# chandle

**chandle** objects are used to streamline communication between mex and MATLAB in iterative processes.

• -

- BlockStrain xxx
- nl\_inout xxx

# chandle

# Purpose

**chandle** objects are used to streamline communication between mex and MATLAB in iterative processes.

Creation generates a C copy of the matlab array and returns a chandle object containing the ID. Register the chandle object for mexAtExit.

• chandle.numType lists currently implemented chandle subtypes.

## DiagNewmark

DiagNewmark is an implementation of the Newmark scheme when assuming a fixed diagonal full Jacobian as occurs in modal domain transients (explicit or implicit) and explicit dynamics.

#### BlockStrain

xxx

#### nl\_inout

xxx support for observation performed in C. .iopt for standard integer options. .N field : Nunl, (c,1),(c,2),cTrans, (b,2),(b,1),bTrans, Nopt[8],Niopt[9],size(unl,3)[

```
.opt field ? tc[1] dt0[2] K[3] Fmax[4] Fu functions currently implemented in C are
```

- FuPower uses .opt=[k n] and  $s_{nl} = k \ u_{nl}^n$ .
- FuTable xxx
- FuExpon xxx  $s_{nl} = a \ b^{u_{nl}}$ .
- Maxw reimplementation of nl\_maxwell.m file
- FuDahlC Dahl model with constant force
- **STS** PSA scalar STS
- CLIMA2 xxx Rosatello
- FuFric basic friction model with  $Fk=K^*u$  bounded by +-Fmax

The header of the associated class is

```
// nl_inout non linearity 1003
class chandleNl_inout: public chandle {
public:
 int *irc, *jcc,*irb, *jcb,*iopt;
 double *prc,*pic,*prb,*pib,*unl,*vnl,*snl,*opt;
 int N[11]; // Nunl, (c,1),(c,2),cTrans, (b,2),(b,1),bTrans, Nopt[7],Niopt[8],size(u
 __Fu Fu;// (*Fu)(chandleNl_inout*,struct _ROr);
mxArray* MexData[2];
 chandleNl_inout();
 ~chandleNl_inout();
 void Residual(struct _ROr ROr, double* fc);
 void initCpt(); // Initialize pointers
 void EndStep(); // propagate internal states using StoreType strategy
};
// Residual structure ------
struct _ROr {
         Nk,Nnl;
  int
  double RayleighM,RayleighK,tc;
  double *u,*v,*a,*FNL;
 };
// Default function handle
typedef void(*__Fu)(chandle* ph, struct _ROr ROr);
```

xxx Dynamic loading of user Non-linearity written in C. Each of this

# Non linearities list

#### Purpose

List of supported non linearities. It is possible to create new ones (sdtweb nl\_fun)

#### nl\_inout

nl\_inout is the more general non linearity, using observation and command matrix.  $f_{NL} = b \times f(C.u, C.v)$ 

The pro.NLdata structure has fields

.type	'nl_inout'
.lab	Label of the non linearity.
.Fu	defines strains based on strains in .unl and strain rates in .vnl. Accepted forms are
	described in section 1.2.
.Fv	The same as .Fu excepted for string predefined laws that define also .Fv.
.unl0	Optional. Define an offset in observation (before applying Fu) as $unl = c * u + unl0$ . It
	can be a vector giving direct offset (as many lines as c). It can also be a string defining
	what offset to apply : the only strategy implemented at this time is unlo='q0' to
	remove observation of the static from observation at each time step. It can also be
	the name of a curve stored in model stack.

The non-linear kinematics can be defined using observation/sensor command/loads as detailed in section **??** or elements section 1.1.3 and following.

By default, no jacobian is computed for this non-linearity. Experimental jacobian are computed according 3 methods according to the NL.Jacobian value:

- 0 : no jacobian. (default).
- 1 : tangent matrices.
- 2 : fixed jacobian (can be max stiffness / damping or mean, ...).

Then computed matrices are then multiplied by NL.alphaJK factor for jacobian stiffness, and NL.alphaJD factor for jacobian damping.

#### nl\_maxwell

nl\_maxwell describes rheological models using stiffness and damping.

.type	'nl_maxwell'
.lab	Label of the non linearity.
.Sens	Observation definition. Cell array of the form {SensType,SensData} where SensType
	is a string defining the sensor type and SensData a matrix with the sensor data (see
	sdtweb sensor).
.Load	data structure defining the command as a load (with .DOF and .def fields).
.SE	superelement that defines the rheological model. Only matrices are used (.K field).
	Mass matrix is ignored. The .DOF field is unused and first DOF are assumed to be
	the observations defined, and following correspond to internal states.
.NLsteps	Number of sub steps for the integration.
.StoreFNL	strategy to store FNL output.
Ncell	number of cells.
<b>T</b> 1	

Jacobian is computed using a Guyan condensation keeping only the observation (internal states are condensed) to obtain tangent damping and stiffness.

Internal states are integrated using an independent finite differences explicit scheme, with the same step of time as the main scheme, or a subsampling NL.NLsteps times.

At the first residue computation, the initial internal states are computed according to initial condition in terms of displacements and velocities through a time integration until variation of speed between the 2 last computed steps is lower than opt.RelTol.

Force on the observation DOF (F), displacement (Qc) and velocity (dQc) of the internal DOF, displacement and velocity observations are stored in the NL output.

The command nl\_spring db Fu" type" is a database of generalized maxwell rheological models. type can be:

• zener standard viscoelastic model. Parameter k0, k1 and c1 can be given as a string of the form db Fu"zener k0 k0 k1 k1 c1 c1" in the command.

The example of the standard viscoelastic model is detailed here as an illustration. The standard viscoelastic model, also known as Zener model, is composed by a spring  $(K_0)$  in parallel with another spring  $(K_1)$  and a serial dashpot  $(C_1)$  as displayed figure 3.5.



Figure 3.5: Standard viscoelastic model.

In the Laplace domain, the relation between the relative load and the relative displacement is given by

$$F(s) = K(s)X(s) = \frac{K_0K_1 + (K_0 + K_1)C_1s}{K_1 + C_1s} = K_0\frac{1 + s/z}{1 + s/p}$$
(3.1)

where p and z are respectively the pole and the zero of the model

$$p = \frac{K_1}{C_1} \tag{3.2}$$

$$z = \frac{K_0 K_1}{(K_0 + K_1) C_1} \tag{3.3}$$

The maximum loss factor is

$$\eta_m = \frac{p-z}{2\sqrt{pz}} = \frac{1}{2} \frac{K_1}{\sqrt{K_0 \left(K_0 + K_1\right)}}$$
(3.4)

and obtained for pulsation

$$\omega_m = \sqrt{pz} = \frac{K_1}{C_1} \sqrt{\frac{K_0}{K_0 + K_1}}$$
(3.5)

 $K_0$  is the static stiffness of the model. Typically  $K_1 = \frac{K_0}{2}$  and  $C_1$  is defined so that the damping is maximal for the frequency of interest.

Following example considers  $K_0 = 1000N/m$ ,  $K_1 = 500N/m$  and  $C_1 = 1.4Ns/m$ . These parameters lead to a maximum loss factor of 20.14% for a frequency of 46.41Hz. The module and the loss factor are represented in figure 3.6.


Figure 3.6: Module and loss factor.

Following example consists in a mass of 1e-2kg linked to the ground by the zener model. Initial displacement corresponding to a 1N load on the mass is imposed and then a time simulation is performed.

```
% parameters
```

```
param.m=1e-2; param.dt=1e-4; param.N=1e3;
param.k0=1e3; param.k1=param.k0/2; param.c1=1.4; % zener parameters
% define model
model=struct('Node',[1 0 0 0 0 0 0],...
'Elt',[Inf abs('mass1') 0; 1 0 0 param.m 0 0 0]);
% define nl_maxwell data
data=nl_maxwell(sprintf('db Fu"zener k0 %.15g k1 %.15g c1 %.15g"',....
param.k0,param.k1,param.c1));
data.Sens{2}=1.03; % translation sensor defining nl_maxwell inputs
% define associated property
r1=p_spring('default'); r1=feutil('rmfield',r1,'name');
```

```
r1.NLdata=data; r1.il(3)=param.k0;
r1.il(1)=100; model=stack_set(model,'pro','zener',r1);
% define option for time integration
opt=d_fetime('TimeOpt');
opt.NeedUVA=[1 1 1];
opt.Follow=1; opt.RelTol=-1e-5;
opt.Opt(7)=-1; % factor type sparse
opt.Opt(4)=param.dt; opt.Opt(5)=param.N; % NSteps
%opt.IterEnd='eval(opt.Residual)'; % to compute real FNL for current state
% Initial state
r1=data.SE.K{3}\[1;0]; r1=r1(1); % initial displacement for 1N load
model=stack_set(model,'curve','q0',struct('def',r1,'DOF',1.03));
% Time computation
def0=fe_time(opt,model); ci=iiplot; % compute
% The same but NL as a model
SE2=data.SE;
SE2.Elt(end+1:end+2,1:6)=[Inf abs('mass1'); 1 0 0 param.m 0 0];
SE2=fe_caseg('assemble -secdof -matdes 2 3 1 -reset',SE2);
r1=SE2.K{3}\[1;0]; %r1=r1(1);
SE2=stack_set(SE2,'curve','q0',struct('def',r1,'DOF',SE2.DOF));
def20=fe_time(opt,SE2); % compute
F20=SE2.K{2}*def20.v+SE2.K{3}*def20.def; F20=F20(1,:);
% zener labs: {'zener-F1', 'zener-q1', 'zener-q1-1', 'zener-dq1', 'zener-dq1-1'}
NL20=struct('X',{{def20.data {'LIN-F1';'LIN-Qc1';'LIN-dQc1';'LIN-unl1';'LIN-vnl1';'ft'}
 'Xlab',{fe_curve('datatypecell','time')},...
 'Y', [F20' (fe_c(def20.DOF,1.03)*def20.def)'...
     (fe_c(def20.DOF,3.03)*def20.def)'...
     (fe_c(def20.DOF,1.03)*def20.v)'...
     (fe_c(def20.DOF,3.03)*def20.v)' zeros(size(def20.def,2),1)
                                                                       1);
NL20.name='NLfromLIN';
iicom('curveinit',{'curve','NL(1)',ci.Stack{'NL(1)'};
        'curve',NL20.name,NL20});
A=ci.Stack{'NL(1)'}.Y(2:end,:);B=NL20.Y(2:end,:);t=NL20.X{1}(2:end);i2=any(A);
if norm(A(:,i2)-B(:,i2),'inf')/norm(B,'inf')>0.01
 figure(1);plot(t,A,'--o',t,B,'-')
 sdtw('_err', 'something has changed')
end
```

#### DofKuva

DofKuva defines a non linear load of the form

 $av_{Dof}^{e}[K] \{V\}$  with a scalar coefficient a, a scalar  $v_{Dof}$  extracted from displacement, velocity or acceleration, and V a field specified as follows

- .type 'DofKuva'
- .lab Label of the non linearity.
- .Dof Dof of Case.DOF.
- .Dofuva [1 0 0] for displacement Dof, [0 1 0] for velocity and [0 0 1] for acceleration.
- .MatTyp Type of the matrix K (see MatType). Desired matrix is automatically assembled before time computation.
- .factor Scalar factor a.
- .exponent Exponent of the DOF.
- .uva Type of vector V:  $[1 \ 0 \ 0]$  for displacement,  $[0 \ 1 \ 0]$  for velocity and  $[0 \ 0 \ 1]$  for acceleration.

For example one can take in account gyroscopic effect in a time computation with a NL of the form

```
model=stack_set(model,'pro','DofKuva1005', ... % gyroscopic effects
    struct('il',[1005 fe_mat('p_spring','SI',1) 0 0 0 0 0],...
    'type','p_spring','NLdata',struct(...
        'type','DofKuva','lab','gyroscopic effect', ...
        'Dof',1.06,'Dofuva',[0 1 0],'MatTyp',7,...
        'factor',-1,'exponent',1,'uva',[0 1 0])));
```

#### DofV

**DofV** defines a non linear effort of the following form (product of a fixed vector and a dof)  $Dof^{exponent}.V$ 

	.type	'DofV'
NDdata fields:	.lab	Label of the non linearity.
	.Dof	Dof of Case.DOF.
	.Dofuva	[1 0 0] for displacement Dof, [0 1 0] for velocity and [0 0 1] for accele
	.exponent	Exponent of the DOF.
	.def	data structure with fields .def which defines vector V and .DOF which defi
		sponding DOF.

#### nlspring

nlspring defines a non linear effort from rheological information (stop, tabulated damping or stiffness laws etc.) between 2 DOF.

To define a non linear spring, one has to add a classic celas element, linear spring between only

2 DOF. The non linear aspect is described by associated properties as a 'pro' entry in the model Stack.

For the moment, the 2 DOF of the non linear spring have to be given in the .dof1 and .dof2 fields of this entry.

One can describe non linearity by a formal rheological description using one or more of following fields in the pro Stack entry:

- .But : [dumax k0 c0 dumin k1 c1]. For du from dumin to dumax, f=0. For du>dumax, k0 stiffness is applied to du-dumax, and for du<dumin, k1 stiffness is applied to du-dumin. Damping is not taken in account at this time (due to tabulated law strategy).
- .Fsec : [fsec,cpenal]. For dv<-fsec/cpenal or for dv>fsec/cpenal, f=fsec is applied. For -fsec/cpenal<dv<fsec/cpenal,f=cpenal\*dv is applied. If omitted, cpenal=1e5.
- .K
- .C

This information will be converted in tabulated laws Fu and Fv using nl\_spring tab (low level call that should be automatically called at the beginning of time computation).

One can also describe non linearity with a tabulated effort / relative displacement and effort / relative velocity law between the DOF (dof2-dof1), respectively in the Fu and Fv fields of the pro Stack entry. First column of Fu (resp. Fv) gives the relative displacements (resp. velocities) and second column gives the efforts. One can give a coefficient av factor of Fv depending on relative displacement as a third column of Fu. It can be useful to describe a non linearity depending on relative displacement and relative velocity. Force applied is F=av(du) . Fv(dv). It is used in particular to describe damping in a stop (.But NL).

Following example performs a non linear time computation on a simple 2-node model:

```
pro={'pro', 'celas1',...
struct('il',[100 fe_mat('p_spring','SI',1) 1e3 1 0 0 0],...
'type', 'p_spring',...
'NLdata', struct('type', 'nlspring',...
'dof1',1.03,'dof2',2.03,...
'but',[1e-3 1e3 0 -4e-4 1e3 0]))};
model=stack_set(model,pro);
```

sdtweb('\_eval', 'd\_fetime.m#Mesh2Dof') % Simple mesh

% perform an implicit NLNewmark simulation % generate fe\_time opt structure

```
opt=d_fetime('TimeOptNLNewmark ts 1e4 dt 1e-3 needUVA 110');
def=fe_time(opt,model);
% perform a variant Newmark explicit simulation
opt=d_fetime('TimeOptExplicit dt 1e-3 ts 1e4 needUVA 110 -fclean"-cf-1"');
d2=fe_time(opt,model);
% plot some comparison between results
figure(10);
subplot(211);plot(def.data,def.def');xlabel('Time [s]');title('displacement')
subplot(212);plot(def.data,def.v');xlabel('Time [s]');title('velocity')
subplot(211);hold on;
plot(d2.data,d2.def','r--');xlabel('Time [s]');title('displacement')
hold off;
subplot(212);hold on;
plot(d2.data,d2.v','r--');xlabel('Time [s]');title('velocity')
legend('Implicit', 'Explicit');
hold off
```

Following example deals with a clamped-free beam, with a bilateral bump stop at the free end. XXX: image

```
% define model:
L=1; b=1e-2; h=2e-2; e=1e-3; % dimensions
model=[];
model.Node=[1 0 0 0 0 0; 2 0 0 0 L 0 0];
model.Elt=[Inf abs('celas') 0 0;
                    100 1 110 0; % linear celas
          2020
           Inf abs('beam1') 0 0;
           1 2 1 1
                      0 1 0 0
           1:
model=feutil(sprintf('RefineBeam %.15g',L/20),model);
model=fe_case(model,'FixDof','base',1); % clamps 1st end
model=fe_case(model,'FixDof','2D',[0.03;0.04;0.05]); % 2D motion
% model properties:
model.pl=m_elastic('dbval 1 steel');
model.il=p_beam(sprintf('dbval 1 BOX %.15g %.15g %.15g %.15g',b,h,e,e));
% Bump stop NL:
model=stack_set(model,'pro','celas1',...
        struct('il',[100 fe_mat('p_spring','SI',1) 1e-9 0 0 0 0],...
```

```
'type', 'p_spring',...
               'NLdata', struct('type', 'nlspring',...
                                'dof1',2.02,'dof2',0,...
                                'but', [0.02 5e2 0 -0.02 5e2 0], ... % gap knl cnl...
                                'umin',3)));
if 1==1
 model=fe_case(model, 'DofLoad', 'in', struct('DOF', 2.02, 'def', 50));
 model=fe_curve(model,'set','input','TestStep t1=0.02');
else
 f=linspace(12,18,3);
 model=fe_case(model,'DofLoad','in',struct('DOF',2.02,'def',1));
model=fe_curve(model,'set','input',sprintf('Testeval cos(%.15g*t)',f(1)*2*pi));
end
model=fe_case(model,'setcurve','in','input');
% Time computation:
opt=d_fetime('TimeOpt dt=1e-3 tend=10'); opt.NeedUVA=[1 1 0];
def=fe_time(opt,model);
```

#### RotCenter

The **Rotcenter** joint is used to introduce a penalized translation link between two nodes A and B (rotation DOFs of NL entry are ignored), where the motion of A is defined in a rotating frame associated with angle  $\theta_A$  and large angle rotation  $R_{LG}(\theta_A)$ . The indices G and L are used to indicate vectors in global and local coordinates respectively.

The positions of nodes are given by

$$\{x_A\}_G = [R_{GL}] (\{p_A\} + \{u_A\}_L) \{x_B\}_G = (\{p_B\} + \{u_B\}_G)$$
(3.6)

which leads to expressions of the loads as

$$\{F_A\}_L = [R_{LG}] \left( K \left( \{x_B\}_G - \{x_A\}_G \right) \right) \{F_B\}_G = K \left( \{x_A\}_G - \{x_B\}_G \right)$$

$$(3.7)$$

To account for viscous damping loads in the joints, one must also compute velocities. Using (??), one obtains

$$\{\dot{x}_A\}_G = [R_{GL}] \left(\{\dot{u}_A\}_L + \{\omega(t)\} \land \{p_A + u_A\}_L\right) \{\dot{x}_B\}_G = \{\dot{u}_B\}_G$$

$$(3.8)$$

Velocity computations are currently incorrect with  $u_A$  ignored in the rotation effect. So that viscous damping loads can be added

$$\{FC_A\}_L = [R_{LG}] \left( K \left( \{\dot{x}_B\}_G - \{\dot{x}_A\}_G \right) \right) \{FC_B\}_G = K \left( \{\dot{x}_A\}_G - \{\dot{x}_B\}_G \right)$$
 (3.9)

For a linearization around a given state (needed for frequency domain computations or building a sensor observation matrix),

$$\left\{ \begin{array}{c} q_{AG} \\ q_{BG} \end{array} \right\} = \left[ \begin{array}{c} R_{GL} & 0 \\ 0 & I \end{array} \right] \left\{ \begin{array}{c} q_{AL} \\ q_{BG} \end{array} \right\}$$
(3.10)

In global basis, stiffness matrix of a celas link is given by

$$k \begin{bmatrix} I & -I \\ -I & I \end{bmatrix}$$
(3.11)

which leads to the following stiffness matrix

$$\begin{bmatrix} R_{GL}^T & 0\\ 0 & I \end{bmatrix} k \begin{bmatrix} I & -I\\ -I & I \end{bmatrix} \begin{bmatrix} R_{GL} & 0\\ 0 & I \end{bmatrix} = k \begin{bmatrix} I & -R_{GL}^T\\ -R_{GL} & I \end{bmatrix}$$
(3.12)

where  $q_A$  DOFs are in the local basis (motion relative to the shaft in its initial position) and  $q_B$  are in the global frame.

data describing this link is stored in model stack as a p\_spring pro entry. Stiffness and damping are stored respectively as 3rd and 5th column of the data.il field (standard linear spring, see sdtweb('p\_spring')).

NDdata fields:

- .type string 'RotCenter'.
- .sel a FindElt command to find celas of RotCenter type.
- .k this field should not be used. .JCoef field should be used instead and has priority. Stiffness used for Jacobian computation. Damping is not taken in account in Jacobian in this case.
- .JCoef coefficient of celas stiffness and damping for jacobian computation. Default is 1.
- .drot the rotation DOF.
- .lab label.

#### nl\_rotCenter

This non linearity can be used to connect 2 points A and B, where the motion of A is defined in a rotating frame associated with angle  $\theta_A$  and large angle rotation  $R_{LG}(\theta_A)$ . More generally A and B are no real nodes but defined implicitly as observation matrices. **nl\_rotcenter** is an extension of **RotCenter** documented above, using observation matrices which is more general.

• .type string 'nl\_rotcenter'.

- .sel a FindElt command to find elements associated to the NL link (XXX really used ?).
- .JCoef coefficient of celas stiffness and damping for jacobian computation. Default is 1.
- .drot the rotation DOF.
- .lab label.
- .Weights (optional) Weight of the stiffness in a pivot link (in fact computed force is multiplied by the weight factors before being applied so that the sum of weight coef divided by number of points by pivot should be equal to 1).
- .Stack Stack of cta coupling. Of the form {'cta', 'name', {r1,r2}}, where 'cta' is a constant string defining the type of the link, 'name' a string containing the name of corresponding links. r1 is the observation in the first (rotating) part. It is a data structure with fields .Node defining the nodes involved, .cta defining the observation matrix, .DOF defining corresponding DOF (as many columns as in .cta) and .SeName defining as a string the name of the superelement where cta is defined (XXX if omitted, it is assumed that DOF and cta are defined on the model.DOF - no superelement -). r2 is the same for the non rotating part.

An example can be found in t\_nlspring 2beam (XXX this must be generalized to make it separated from psa08).

Temporary strategy was to define prior a RotCenter link, then convert it to the nl\_rotcenter link using model=psa08('nl\_rotcenterFix',cf.mdl,RO); command. This should no longer be used XXX.

Default uses the damping and stiffness defined in the il field of the p\_spring pro entry to model a linear spring/damper between the 2 parts (stiffness il(3) and damping il(5)).

Defining a  $\mathbf{x}\mathbf{b}$  parameter, the Excite NONL law will be applied instead of the spring/damper. Parameter that are to be defined are

- .xb Radial clearance.
- .kb Stiffness at radial clearance.
- .cb Damping at radial clearance.

Stiffness and damping at initial position are given in corresponding  $p\_spring$  properties il(3) and il(5). For example:

cf.mdl=nl\_spring('setpro ProId 103 k 371 c 2000e-3 xb 0.03 kb 37100 cb 5', cf.mdl); One can also use a squeeze-film type law defining a .muRL3 field, with following parameters (XXX this need to be reactivated and updated):

• .muRL3 oil viscosity\*bearing radius\*bearing length<sup>3</sup>.

- .boundary Boundary condition (0=Sommerfeld,1=Gumbel(p;0)).
- .N\_theta Number of integration point.

Defining a .RelF field, one can define a sub call to another non linearity to compute the force resulting from the eccentricity computed by nl\_rotcenter. In that case, the .RelF field contains the NLdata field of the sub NL to be called. Following example (XXX need to generalize psa08 ShaftRebuild and ShaftTime XXX) define a zener link in the bearing of simple 2beam example:

```
cf=feplot(2);beam1('beam1In',1); t_nlspring('2beam-build');%old inertia
psa08('ShaftRebuild',cf) % define large rotation DOF
psa08('ShaftTime 2000 1 1e-5 0',cf) % Omega(rpm), N_rot, dt, theta0
if beam1('beam1In')~=1;error('clear problem');end
mo1.il(2,6)=0; % No gyroscopic because of shaft beam1
opt.RotFollow=1;% Follow velocity and not iterations
r1=stack_get(mo1,'pro','celas103','GetData');
% define zener
NLdata=nl_maxwell('db Fu"zener k0 125e3 k1 50e3 c1 0.4"');
r1.NLdata.RelF.type='nl_maxwell'; r1.NLdata.RelF.NLdata=NLdata;
mo1=stack_set(mo1,'pro','celas103',r1);
def=fe_time(opt,mo1);
```

#### rod1

XXXEB : following needs to be checked The rod1 non-linear connection is a simple penalized rigid link. One considers two nodes A and B (see figure 3.7).



Figure 3.7: Large rotation rod functional representation.

Currently, one can introduce masses at points A and B. mass2 elements should be used to account for the actual position of the center of gravity.

#### Non linearities list

The global non linear load associated with the rod is thus

$$F_{rod} = k_r \left( \| \{ x_B - x_A \} \| - L_0 \right) \frac{\{ x_B - x_A \}}{\| \{ x_B - x_A \} \|}$$
(3.13)

which accounts for a load proportional to the length fluctuation around  $L_0$  (penalized rod model). When linearizing, one considers a strain energy given by  $k_r ||q_B - q_{A'}||^2$  with the motion at node A' being related to the 6 DOFs at node A by

$$\{q_{A'}\} = \begin{bmatrix} I & \left[\vec{AB}\wedge\right]\\ 0 & I \end{bmatrix} \{q_A\}$$
(3.14)

Node A node is free to rotate. The linearized stiffness thus corresponds to an axial stiffness in the direction of the rod. The computation of the stiffness is however based on the current position of the extremity nodes, a difficulty in model manipulations is thus to translate these nodes.

data describing this link is stored in model stack as a p\_spring pro entry. Stiffness and damping are stored respectively as 3rd and 5th column of the data.il field (standard linear spring, see sdtweb('p\_spring')). NL information is stored in the data.NLdata field which has itself following fields :

- type : string 'rod1'.
- sel: a FindElt command to find associated celas of rod1 type (('proid100').
- ulim : build tabulated law from -ulim to ulim. Default is 1e3.
- lab : label.

nl\_gapcyl



This non-linearity implements non-linear contact between two cylinders of radius  $R_O$  for the outer cylinder and  $R_I$  for the inner cylinder with motion defined on the cylinder center line. Assuming

the mesh to be defined in an updated Lagrangian configuration where the center lines  $X_I$  and  $X_O$ at not coincident, the positions in a deformed states are given by  $x_I = X_I + u_I$  and  $x_O = X_O + u_O$ . Contact may only occur when the cylinders are not centered. When the two cylinders are not centered the non-linear observation is given by

$$u_{NL} = x_O - x_I = u_O - u_I + (X_O - X_I) = [c] \{q\} + u_{NL0}$$

From this distance between the center lines, a cylindrical basis is defined with  $e_r(q)$  along the direction from  $x_I$  to  $x_O$  and  $e_\theta$  forming a direct basis with the cylinder axis (kept constant from the initial value of the updated lagrangian position).

The functional definition of the contact force uses the gap in the  $e_r$  direction defined by

$$g = \{e_r\}^T \{u_{NL}\} - (R_O - R_I)$$

where  $R_O - R_I = d$  is stored as parameter NLdata.d and the contact leads to two opposite forces on the cylinder center lines

$$\{f_I\} = \{-f_O\} = f(g)\{e_r\}$$

xxx account for initial contact pressure in the updated configuration xxx

The current implementation assumes rotations to be small enough to ignore the difference between  $e_r$  and the corresponding vector  $E_r$  in the upated Lagrangian configuration.

xxx if update of mesh leads to lateral slip, then  $u_I$  may account for longitudinal position of the contact point along the beam using shapes functions.

When linearizing the contact around a given point, the stiffness  $\partial f \partial g$  only occurs in the  $e_r$  direction.

#### nl\_modaldmp

Implementation of modal damping. Although modal damping is not a non-linear feature in itself, its implementation requires it to be declared as a non-linearity.

The concept is to provide shapes defined on a part of a model with associated damping ratios. nl\_modaldmp handles the kinematic projection on the model which can contain superelements. In the case where superelements are used and concerned with modal damping, the shapes provided must be written on the physical DOF of the superelements.

The set of shapes must be stacked in model with a valid ID field. It is a common deformation SDT data structure (see sdtweb def), with an additional .ID field. The .data field is equivalent to the ones of complex modes (see fe\_ceig). It is a matrix of two columns respectively giving the frequency and the target damping ratio for each mode.

Since modal damping implies a modal sensor, the features performs both by default. It is however possible to simplify it as a pure modal sensor. The theory around modal sensing/damping can be found in [4].

The pro.NLdata structure has fields

• type: string 'nl\_modaldmp'.

- CurveId: the curve ID stacked in model which provides the shapes and their damping ratios.
- SensorOnly: to use the feature only as a modal sensor in a def data structure.

The NLdata structure generation can be integrated using an nl\_modaldmp('db') call. See sdtweb nl\_spring#setpro for this integration. This is used in transient simulations, and in complex mode computations, see nl\_solve.

#### $nl_contact$

See p\_contact, ctc\_utils.

# Creating a new non linearity: nl\_fun.m \_\_\_\_

#### Purpose

The structure of nl\_spring allows creating any new non-linearity through the use of a dedicated function, named nl\_fun.m. This function which non-linearity name will be fun, will be automatically called by nl\_spring for classical operations.

The function structure has been designed to comply with specific needs. Standard calls have been defined, which are detailed below:

• Residue computation, called by mkl\_utils (sdtweb mkl\_utils), must output the entry force minus the non linear force computed. The call performed is

nl\_fun(r2,fc,model,u,v,a,opt,Case)

This call is low level and must modify fc using sp\_util('setinput') as fc-fnl where fnl is the non linear force computed. Note that this is the only possible call for nargin==8. Note that mkl\_utils allows a formalism with precomputed observations, using fields unl.

• Jacobian computation, must output the tangent stiffness and tangent damping matrices associated to the non linearity. The call performed is

[kj2,cj2]=nl\_fun(NL,[],model,u,v,[],opt,Case,RunOpt);

This call must output either empty matrices if no tangent nor Jacobian matrix is associated to the non linearity, or matrices expressed on the DOF vector of Case.DOF. The first matrix is the tangent stiffness matrix, the second one is the tangent damping matrix. Typically there are 3 normalized methods to be defined (but not all of them must be defined, and more can be defined) according to the NL.Jacobian value:

- 0 : no jacobian. (default).
- -1: tangent matrices.
- 2 : fixed jacobian (can be max stiffness / damping or mean, ...).

Then computed matrices are then multiplied by NL.alphaJK factor for jacobian stiffness, and NL.alphaJD factor for jacobian damping.

• Initializations for fe\_time, must initialize the model non-linearity for non linear forces computation

The call must generate the non linearity stored in model.NL, it can optionally generate non linear DOF and labels. The call performed is of the type.

NL=nl\_fun('init',data,mo1);

NL is a struct containing at least the field type with the nl\_fun handle (*e.g.* NL.type=@nl\_fun). data contains the Stack, pro entry, and mo1 is the model, named mo1 where the call is performed.

• **ParamEdit** returns the **ParamEdit** string allowing integrated parameters interpretation (for internal SDT use).

The call performed is of the type.

st=nl\_fun('ParamEdit');

• db returns default NLdata fields for a non linearity. This allows integrated building of nonlinearities in a model. This function can call ParamEdit to allow interactive setup.

This call must return a NLdata field and is of the type

NLdata=nl\_fun('db data 0');

• Energy post treatments capability, should return the elastic energy stored in the nonlinearity as a vector with as many lines as time steps in the output.

The call performed by nl\_solvePost is of the form

```
r2 = nl_fun('PostEnerNL');
```

The non linearity function can access in caller fields RO, out, model, NL, i1 with

- RO a structure with fields <code>EnerP</code> and <code>EnerK</code> respectively containing the potential and kinetic energy.
- out the fe\_timeoutput.
- model the model used in the simulation.
- NL the NL containing the data of the non-linearity called.
- i1 the row index of out.Post that is currently generated.
- **Renumbering** capability, must return the non-linearity written for the new renumbered nodes, elements, dof, . . .

The call performed (by feutilbfor example) is of the type

NL=nl\_fun('renumber',NL,nind);

**nind** is the renumbering vector.

The designed nl\_fun template is given in the non-linear toolbox, sdtweb nl\_fun.m#1. It is a functional non linear function, computing a zero non linear force. The definition of a non linearity using nl\_fun in a standard SDT model is given in the following.

```
% A standard SDT model
model=struct('Node', [1 0 0 0 0 0 0; 2 0 0 0 0 1],...
             'Elt', [Inf abs('celas') 0 0;
             1 2 3 -3 0 1 0 10; % linear celas
            ]);
% Define a non linearity of type nl_fun
model=nl_spring('SetPro ProId 100',model,nl_fun('db data0'));
%Equivalent to
% model=stack_set(model,'pro','nl_fun',...
% struct('il', [100 fe_mat('p_spring', 'SI', 1)],...
%
         'type', 'p_spring',...
         'NLdata',struct('type','nl_fun','data',[])));
%
% Define the case
model=fe_case(model, 'FixDof', 'base', 1);
model=fe_case(model,'DofLoad','in',struct('DOF',2.03,'def',1));
model=fe_curve(model,'set','input','TestStep t1=0.02');
model=fe_case(model,'setcurve','in','input');
% Define the TimeOpt and compute the solution
opt=nl_solve('TimeOpt'); opt.Opt([4 5])=[1e-3 1e4];opt.NeedUVA=[1 1 0];
def=fe_time(opt,model);
```

# nl\_solve

## Purpose

Integrated non linear simulations

# Description

The simulation of non linearities require special handling in SDT, which is packaged in the non linear toolbox. This function aims at performing classical studies, such as done by fe\_simulfor classical SDT models with this special handling.

See **nllist** for the list of supported non linearities.

## TimeOpt

nl\_solve('TimeOptMethod',RO) used to initialize fe\_time options for later simulation. Currently
implemented methods

- Explicit Newmark scheme
- Stat non-linear static Newton
- Theta method time integration.
- ModalNewmark uses an optimized fully C based integration for the case where DOF correspond to modal degree of freedom. The stepped sine strategy is discussed in section 2.3.2.
- NLNewmark default implicit Newmark scheme.

Associated options provided in  ${\tt RO}$  or in the command are

• .tend end time of simulation. Used to initialize .ts=ceil(tend/dt).

#### Static

To compute the static state of a model with non-linearities.

q0=nl\_solve('static',model);

It is possible to use custom fe\_time simulation properties using the model stack entry info, TimeOptStat. See nl\_spring TimeOpt for fields and defaults.

It is possible to use as command option any field from the usual static simulation option, see sdtweb nl\_spring#TimeOpt to have more details. *E.g.* To redefine on the fly the maximum number of iteration, one can enter [q0,opt]=nl\_solve('static maxiter 100',model);.

By default, the staticNewton algorithm implemented in fe\_time is called.

An Uzawa algorithm is also implemented in nl\_solve, under the method static nl\_solve uzawa. This algorithm is very different from the staticNewton one since here the solution is not incremented

but fully re-computed at each iteration. This is useful when some non-linear forces do not derive from potentials. Command StaticUzawa can be used in nl\_solve – to access it: q0=nl\_solve('static Uzawa', model);.

#### Mode

The definition of modes for non-linear models is not straight forward. This command aims at computing tangent modes as function of a non-linear model current state. The resolution thus concerns a linear model with tangent stiffness, damping matrices corresponding to the model current displacement, velocity, acceleration state. The eigenvalue solvers used are then fe\_eigfor real modes and fe\_ceigfor complex modes.

By default, modes tangent to a static state are computed. A static simulation is performed to produce a model state from which tangent matrices are computed. It is also possible to compute tangent modes at specific instants during a transient simulation, at SaveTimes instant, and to store frequency/damping data and deformations.

A set of command options allows precizing the mode computation wanted and the output. Accepted command options to control the model computation itself are

- -allmatdes to ask for an assembly with all matrix types assembled, the default assembly command used is -matdes 2 3 1. This command can be used to keep specific matrix types defined in pre-assembled superelements.
- cpx for complex mode computation (default is real mode computation).
- -evalFNL (in combination with command traj) asks to recompute the FNL field on the fly based on displacements prior to mode computation. This command is useful when solutions used for the tangent state have been imported from an external solver.
- skip skips fe\_timesimulations and performs the complex mode computation based on the zero deformation and with initialized values of non linearities. The behavior will thus depend on the non linearity initialization strategy. *E.g.* for contact see (p\_contact), the -skip option will consider a full contact state.
- stat for mode computation based on a static state (typically after a fe\_time staticNewton simulation). Uses model stack entry info,TimeOptStat.
- time for mode computations during a transient simulation (exclusive with the default -stat option). Uses model stack entry info,TimeOpt.
- traj for mode computations based on states provided as an additional argument.

The -stat and -time options are mutually exclusive and define the base solver options to be used by fe\_timefor the preliminary state computation. With -stat option (default) the stack entry info,

TimeOptStat will be seeked and used if found. With -time option, the stack entry info,TimeOpt will be used if found.

The -traj option is complementary and is used to force the complex mode computation on provided states. On can either provide the state in deformation curve format, see sdtweb def as a last argument, or use predefined stack entries. In -stat mode (default), the model stack entry curve,q0 will be seeked and used if found, if not the result will use the -skip mode. In -time mode, the model stack entry curve,TSIM will be seeked and used. If not found an error will occur. Accepted command options to control the output format are

- -addedOnly (in combination with backTgtMdl) only outputs the tangent matrices as a superelement that would have been added to the base matrices for the mode computation.
- -alpha (requires -cpx) to also output the real mode participation to the complex modes. This is in fact the projection of the complex modes on the real mode basis.
- -backTgtMdl outputs the tangent model that would have been used for mode computation.
- -dataOnly to save only the frequency, damping data (does not store the deformation field). The output is then under a frequency tracking curve in the iiplotformat.
- -fullDOF to output the deformation fields restituted on the unconstrained DOF.
- -keepTval (requires -cpx) to allow keeping the underlying real mode basis when computing complex modes. With val set to 1, the initial real mode basis will be kept under field def.T, as an additional independent output, coherent with the -alpha command option. With val set to 2, the complex modes will not be restituted but expressed on the subspace used for their computation, the subspace basis will be output in def.Mode.TR, allowing a complete compatibility with feploton-the-fly restitution strategy for display. This latter option is the most complete and efficient strategy. Complete subspace information is kept and can be used for further exploitation, complex mode projection on real mode (-alpha) is naturally obtained, and memory footprint is optimized as the storage size of the subspace is commonly lower by a factor 1.5 to 2 than the complex mode basis;
- -noPost is used to skip any solution post treatment, and outputs the raw mode structure straight from the solver.
- -PostFcn'' cam'' is used to perform specific post-treatments on the mode output after computation.
- -real "ModeBas" (requires -cpx) to specify a particular real mode basis on which the complex modes will be computed. The real mode basis is supposed to be stored in the model stack entry curve, ModeBas.

Internally, the solver defines and uses the model stack entry info,SolveOpt structure to handle the options documented above. One can define it as a structure with the fields documented (case sensitive) and provide it instead of the EigOpt input. Additional advanced field are then accessible

- EigOpt a vector providing eigenvalue computation options following the fe\_eigformat.
- cpx'' command'' to externalize the mode computation. This command is by default a boolean telling the solver whether to perform a complex mode computation (set to 1) or a real mode computation (set to 0). If a string is provided, the solver will evaluate it as an external command instead of performing mode computation. One then gets access to the nl\_solve mode computation framework for ones' own solver.
- ind provides a vector of indices that will be used to restrict the output to the indexed modes.
- SubDef provides a command that will be evaluated to perform a dynamic user defined restriction to the output modes, it is thus more general than the ind option. The result of the command has to be a vector of indices.
- AssembleCall to force a specific AssembleCall strategy.

The various input and output strategies allow for the support of several input syntaxes. The following calls are thus accepted, with model a standard SDT model, Case a standard SDT case structure, eigopt either a vector providing options for feleigor a structure with optional fields defined above, def a standard SDT deformation field structure used by -traj when necessary.

```
nl_solve('mode',model);
nl_solve('mode',model,eigopt);
nl_solve('mode',model,Case,eigopt);
nl_solve('mode',model,def);
nl_solve('mode',model,Case,def);
nl_solve('mode',model,eigopt,def);
nl_solve('mode',model,Case,eigopt,def);
```

Sample calls using command options to extract tangent modes are given below.

```
def0=nl_solve('Mode',model)
def0=nl_solve('Mode',model,[5 20 1e3]) % with eigopt
def0=nl_solve('Mode-stat-fullDOF',model);
defT=nl_solve('Mode-time',model);
hist=nl_solve('Mode-time-dataOnly',model);
histC=nl_solve('Mode-cpx-time-dataOnly',model);
defC=nl_solve('Mode-cpx-time-alpha-real''MyBas''-fullDOF',model);
def1=nl_solve('Mode-skip-fullDOF',model);
```

#### Post

The Post command allows performing energy and potential further post treatments of a non-linear simulation. The output is integrated in the standard fe\_timesimulation outputs in field out.Post that is a three columns cell array directly compatible with the iiplot format.

To obtain the post treatments, one must define them prior to starting the simulation. Direct computation of the post-treatments *a posteriori* is also possible.

• Command PostDefine adapts the TimeOpt structure to initialize fields in the output and trigger post treatments in the final cleanup phase. The PostDefine call must thus be performed after the TimeOpt call. Using this command itself prior to a time simulation is enough to obtain the post treatments.

```
opt=nl_solve('PostDefine keys',opt); % adapts the opt structure.
model=nl_solve('PostDefine keys',model); % adapts the opt structure contained in mediate
```

- Command PostLab provides the list of available post treatment keywords. The input is a structure with fields the post treatment keywords and a logical.
- Command PostHist provides an *iiplot curve* structure adapted to the post treatments. On can provide a PostLab structure with fields assigned to 1 for desired posts to obtain the corresponding curve.
- Command PostCompute computes the post treatments and store them in out.Post. This command is internally called if the PostDefine command was used prior to the time simulation. For *a posteriori* computations, the user must provide the out as a standard fe\_time format initialized with Post field and the assembled model. The model must feature a stack entry info, OutputOptions with field Post containing the PostLab structure.

```
% Generate a TimeOpt
opt=nl_spring('TimeOpt');
Perform the time simulation
def=fe_time(opt,model);
% Initialize for post treatments
[def,RO]=nl_solve('PostInit EnerM',def);
model=stack_set(model,'info','OutputOptions',...
struct('Post',RO));
% Assemble model with non linearities
model=fe_case(opt.AssembleCall,model);
% Compute post treatments
def=nl_solve('PostCompute',def,model);
% display in iiplot
iiplot(def.Post);
```

• Command PostInit is an internal function that initializes the output Post field at the start of the simulation. Early initialization is useful if the post treatments are performed on the fly by the OutputFcn.

The following post treatments are available

- EnerP The linear potential, or strain energy.
- EnerK The kinetic energy.
- EnerNL The elastic or strain energy stored in the non linearities.
- EnerM The mechanical energy, defined as EnerP + EnerK + EnerNL.
- PDiss The instant dissipated power.
- EnerDiss The cumulated dissipated energy over time.

Command PostEstimate allows analyzing the energy curves to compute

- Fest an estimation of the vibration frequency (based on quasi-sinusoidal oscillations)
- DmpR an estimation of the damping ratio based on the estimated frequency by computing the dissipated mechanical energy.  $\zeta = \frac{1}{4\pi} \log \frac{E_m(t_0)}{E_m(t_1)}$
- Emax the maximum mechanical energy identified on the cycle analyzed.
- EDiss the dissipated mechanical energy over the cycle analyzed.

The following command options allow altering the estimation

- -cfi to specify the iiplot figure with handle i.
- -bandpass fmax to perform a bandpass from 0 to fmax Hz filtering prior to the analysis.
- -curveName'' name'' to provide the iiplot stacked curve name to exploit.
- -baseOn''name'' to specify on which post treated curve the frequency estimation is made.
- -globalMaxTolval to provide a relative tolerance over which a point is detected as close to the global maximum. This is exploited to detect the peaks over the energy signal analyzed.
- -localMax to estimate the frequency by detecting the zeros of the signal derivative (less robust).
- -unit'' II'' to provide an output unit system.

• XFcn''str'' to provide a function call to be evaluated that can perform further post treatments( e.g. model specific posts). The called function can access out, outLab, st, j1 with out a matrix containing the output with as many lines as provided curves and as many columns as outputs data, outLab a cell array containing the labels of each column, st the curve list (either names or the curves themselves), j1 the curve currently treated.

```
r1 = nl_solve('PostEstimate',def);
r1 = nl_solve('PostEstimate',def.Post{1,3});
r1 = nl_solve('PostEstimate',{'disp(1)'});
r1 = nl_solve('PostEstimate',{'Post_NLsolve(1)'});
```

#### SineSweep

XXX

#### TgtMdlBuild, Assemble

Integrated command to generate linearized models around a specific working point. This command packages the tangent model generation procedures of nl\_solve Mode-backTgtMdl

- TgtMdlAssemble command outputs a fully linearized assembled model, based on the static state provided.
- TgtMdlBuild command generates a linearized model with superelement coupling containing the tangent stiffness and damping contributions of all non-linearities. The following command options are supported
  - -keepName allows naming the superelements with the non-linearity name.
  - -evalFNL forces recomputation of non-linearities states before generation.
  - -staticInterp generates a tangent model allowing tangent matrix interpolation between different static states. The procedure requires the definition of parameters and a method to compute static states. Static states for MinMax configurations of each parameters is then performed. Matrices showing differences as function of parameters are kept and an interpolation rule is defined using the linear finite element functions of a 2<sup>npar</sup> vertices hypercube. The output model has stack fields curve,q0 the series of static states with q0.data providing the parameter points, and info,sCoef providing interpolation rules for each matrix.

```
RA=struct('par',Ra,'q0cbk',{{@my_fun,'ComputeStatic'}});
mo1=nl_solve('TgtMdlBuild-staticInterp',model,RA);
```

Ra is either a Range structure or the content of Range.param (see sdtweb fe\_range), qOcbk is a callback in cell-array format.

```
% Linearized model generation
% sample model with cubes in contact
model=d_contact('cubes cbuild');
% resolve static state
q0=nl_solve('static',model);
% linearized model
mo1=nl_solve('TgtMdlBuild',stack_set(model,'curve','q0',q0));
% check the result
feutil('info',mo1)
SE=stack_get(mo1,'SE'); SE{1,3}
```

# $nl_mesh$

#### Purpose

 $Integrated \ mesh \ modifications \ and \ case \ handling \ for \ non-linear \ applications$ 

### Description

Integrated case handling for constraint penalization and coupling component splitting hare implemented in this function.

Some non-linearities require surface/volume remeshing (*e.g.* definition of conforming interfaces for contact) or adaptations (*generation of thin interface layers*). This function regroups such functionalities. Mesh generation are performed by fe\_gmsh(interface to gmsh) and fe\_tetgen (interface to tetgen, see help fe\_tetgen).

#### Conform

The Conform call is an integrated call to generate conforming meshes between two facing interfaces. The command generates a conforming surface mesh of the face to replace, merges it with the conform mesh of the second interface, replaces the model face mesh and remeshes the model volume to yield a new equivalent volume with a conform face mesh.

```
mo1=nl_mesh('conform eltsel"FindElt"',model,sel);
% sel={eltSelToReplace eltSelForReplacement;...}
```

model is a standard SDT model. **sel** is a cell array containing in each line two FindElt commands specifying the element selection face to remesh and the element selection face to use for the conforming interface for replacement.

- Command option eltsel allows specifying in a string a FindElt command restraining the working area in the original model.
- Command option **smartSize** allows generating a conforming mesh with a coherent mesh characteristic length.
- Command option gmsh allows using gmsh to mesh the final volume.
- Command option tetgen allows using tetgen to mesh the final volume (by default).
- Command option output asks to output the generated mesh in a .mat file.
- Command option OrigContour asks to keep original positions of mid-nodes of the quadratic faces delimiting the volume to remesh. This may however yield mesh wrapping problems when the face to remesh is much coarser than the mesh trace to place for conformity.

- Command option mergeTo allows specifying a FindElt selection command in a string to replace the mesh on another model selection than the one used to generate the conforming interface (which uses eltsel.
- It is also possible to provide additional arguments, which will be passed the the nl\_meshcover call performed in the procedure.

Limitations: The Conform call only supports generation of conforming interfaces when one interface contour fully contains the other interface contour. Handling of more complex contour configurations has not been implemented. Besides, this function has been designed to handle planar surfaces. Additional operation to work on non planar surfaces are left to the user (*e.g.* pre/post projections of the surfaces on a plane).

#### Contour

Call ContourFrom generates SDT beam1or beam3 contour models for CAD definitions. All formats readable by gmsh can theoretically be used. Only the .geo, .stp and .igs are tested. Since .geo files can contain geometric yet undiscretized objects, a 1D meshing pass is performed with gmsh to provide an SDT contour model. This is not supported for other file types.

model=nl\_mesh('contourFrom','file.stp'); % not specifying the type

Call Contour generates an SDT face mesh from an SDT beam1or beam3 contour.

model=nl\_mesh('contour',model);

model is an SDT beam model defining a closed contour.

- Command option lcval allows specifying a characteristic length for gmsh.
- Command option lcminval allows specifying a minimal characteristic length for gmsh.
- Command option quad allows generating quadratic meshes.
- Command option keepNode asks to keep the original contour NodeId for the contour command.
- Command option diag asks to output the gmsh log file for diagnostic problems.
- Command option single tells nl\_mesh that a single contour is defined. This is useful when several closed contours are defined since it is impossible to automatically decide whether each contour is independent or if they define a single complex contour.

- Command option groupval is used in combination to the single command option. This allows specifying which contour group will be meshed, while other possible contours will define holes.
- Command option algo ''val'' allows specifying which algorithm gmsh must use (this depends on the gmsh version, report to the gmsh documentation for more details).
- Command option AllowContourMod allows gmsh adding nodes on the contour provided. By default gmsh is forced not to add nodes to the lines defining the contour to mesh.

#### Cover

The Cover call is designed to mesh the interstice between two closed planar contours, when one fully contains the other. The call is performed as

[newModel,opt,largeContour]=nl\_mesh('cover',model,{eltsel\_large,eltsel\_small});

model is a standard SDT model. Variables eltsel\_large and eltsel\_small are FindElt calls defining the element selection of the respectively large surface and small surface (the small being contained in the large).

The output **newModel** is the mesh generated from the surface contours.

opt outputs additional information about the mesh generation, it is a struct containing fields .NodeAdd specifying the potential nodes added in the interstice space meshed,.nodeEdgeSel1 specifying the NodeId of the nodes located on the eltsel\_large contour, .nodeEdgeSel2 specifying the NodeId of the nodes located on the eltsel\_small contour, and .tname the name of the temporary file containing the generated mesh.

largeContour provides the original contour in beam elements of the eltsel\_large selection.

The following command options are available

- merge allows merging the interstice mesh with the inner mesh of the eltsel\_small selection.
- quad allows generating proper quadratic meshes.
- **smartSize** allows generating an interstice mesh with a characteristic length in coherence with the contour mesh length.
- lcval allows setting the characteristic length to Val to the interstice mesher.
- algo ''name'' allows specifying the meshing algorithm name to the gmsh mesher. See the gmsh documentation for more information.

## Hole[,Groups,Diff,Drill,Gen]

The Hole command series aims at handling hole detection on surfaces and bore drilling generation. The following functionalities are available

Command HoleGroups detects holes on a closed surface and outputs a contour model with element groups relative to each isolated contour. A second output provides the GroupId corresponding to detected holes.

Command HoleDiff provides surface elements that are inside the holes of a given contour. You should better exploit lsutilto get a robust result.

Command HoleGen generates a planar surface with a ruled mesh featuring a hole and controlled radial positions.

- .len length of plate
- .wid width of plate
- .rAnulus radii for base positions
- .ND angular refinement
- .NRext external to bolt radius refinement
- .MatId assign mat/pro id to meshed part
- .noExt remove exterior side
- .Center
- .normal

Command HoleDrill generates cylindrical drills in a model with the possibility to integrate a ruled bolt mesh.

#### Replace

The call **Replace** is designed to replace parts of a model mesh with new given meshes, mesh parts conformity is assumed. It is performed as

model=nl\_mesh('replace',model,nodesToReplace,NewModel,nodeIDtoKeep)

model is a standard SDT model. nodesToReplace is a cell array containing vectors of NodeId specifying the areas to be replaced. NewModel is a cell array containing the new models which will be merged to the mesh in coherence with the removed elements (specified by nodesToReplace). nodeIDtoKeep is an optional argument specifying NodeId of the original model for nodes whose NodeId must not change in the transformation.

#### $nl_mesh$

Control of nodeIDtoKeep per NewModel part is possible by providing a cell array of NodeId list of the same size than NewModel.

The following command options are available

- setMat allows defining a specific MatId to the output mesh.
- setPro allows defining a specific ProId to the output mesh.
- eltset*FindEltString* can be provided to provide an element selection for MatId and ProId assignment.
- keepNoCheck in combination with the use of a third argument nodeIDtoKeep assumes the nodes numbering is correct and forces the nodes original numbering without check.
- -jAll asks to join all elements per type, then separated by MatId
- -inSet asks to maintain coherence with EltId sets. EltId sets for which the totality of a given removed part belonged to will be updated to contain the EltId of the replacement mesh.

#### Rivet

This command generates rivet drills in a specified contour. A model containing a beam contour can be provided, or an EltSel string generating a surface selection (see section and the selface option) on a bigger model. A data structure providing the origins, and rivet radiuses and washer (or rivet head radiuses). The mesh generated between both radiuses is structured. The data structure must contain fields

- Orig providing the rivet centers in an [x y z;...] matrix.
- **radHole** providing the rivet hole radius, either a scalar if all rivets have the same radius, or a line vector providing each rivet radius separately.
- radWash providing the rivet washer (or head) radius, either a scalar if all rivets have the same washer radius, or a line vector providing each rivet washer radius separately.

and can optionally contain fields

- plane To directly provide the contour plane normal to define the drilling, in an [nx ny nz; ...] matrix.
- Ns To define the number of mesh segments in the rivet to washer radius area (default 10), either a scalar if all rivet heads have the same properties, or a line vector defining the property for each rivet separately.

- Nr To define the number of mesh radial nodes in the rivet to washer radius area (default 2), either a scalar if all rivet heads have the same properties, or a line vector defining the property for each rivet separately.
- Command option MatId val allows setting the modified mesh MatId to val.
- Command option ProId val allows setting the modified mesh ProId to val.
- Command option –fill outputs in second argument a compatible mesh of the rivet bores.
- Command option –allQuad outputs the remeshed model with elements only.

Following example meshes a rectangular contour with a few rivet drilling inside.

```
% Generate a global contour
model=struct('Node',[...
1 0 0 0 0 0;
2 0 0 0 10 0 0;
3 0 0 0 10 2 0
4 0 0 0 0 2 0], 'Elt',[]);
model.Elt=feutil('ObjectBeamLine 1 2 0 2 3 0 3 4 0 4 1',model);
model=feutil('refinebeam .2',model);
%feplot(model)
% define rivet positions, eventually planes
RD=struct('Orig',[ 3 1 0;6 1 0;9 1 0],...
'radHole',[.2;.2;.2],...
'radHole',[.8;.8;.8]);
model=nl_mesh('Rivet',model,R0);
cf=feplot(model);
```

GmshVol

This call integrates the generation of a volume mesh from a face mesh with gmsh.

model=nl\_mesh('GMSHvol',model);

model is a standard SDT face mesh model.

• Command option setmat allows specifying a specific MatId to the output mesh.

- Command option setpro allows specifying a specific ProId to the output mesh.
- Command option keepFaces asks to keep original NodeId of the nodes located on the face mesh.
- Command option lc specifies a characteristic length for gmsh.
- Command option clmin specifies a minimal mesh length for gmsh.
- Command option clmax specifies a maximal mesh length for gmsh.

#### ExtrudeLayer

This command generates a non trivial extrusion of a face mesh following the face normal at each node, to generate a volume layer.

model=nl\_mesh('ExtrudeLayer thick Val',model);

model is an SDT model with shell elements (a surface definition).

Command option thick specifies the extrusion thickness. Command option setmat allows specifying a specific MatId to the output. Command option setpro allows specifying a specific ProId to the output.

#### StackClean

This call cleans up a model stack when mesh modifications have been performed. It cleans up stack entries definition that became incoherent with some mesh modifications.

```
model=nl_mesh('StackClean',model);
```

Command option **rmuns** removes stack entries that could not be sorted out. Command option **rmmod** removes stack entries affected by the model modifications.

See also celas, p\_spring, fe\_gmsh

# spfmex\_utils \_

# Purpose

## OfactOptim

This command can be used to set spfmex parameters in order to optimize computation speed for factorization and / or solving. spfmex\_utils('OfactOptim',ki,R0,ofact(1,'lu'));

 $\tt ki$  is the matrix that is used for the optimization.  $\tt RO$  is a data structure defining options with following fields:

- .nCompt Number of computation for result averaging.
- .maxDomain Max size of blocks of the elimination tree (fraction of matrix size).
- .maxZeros Max number of zeros in the blocks of the resolution tree (fraction of matrix size).
- .refineStep Number of step to refine the optimal parameter pair found in the first step. Command option -refine must be added to perform the refine step.

The last argument ofact(1,'lu') is needed in order to call directly spfmex\_utils. Available command options are

- -setopt use default RO.
- -refine performs refine step for optimal search.
- fact to benchmark factorization step.
- solve to benchmark resolution step.
- -plot to plot history in iiplot (xxx this need to be generalized, call to gvdr\_utils plot xxx).

Following example optimize only solving:

spfmex\_utils \_\_\_\_\_

# Bibliography

- G. Vermot Des Roches, Frequency and Time Simulation of Squeal Instabilities. Application to the Design of Industrial Automotive Brakes. Ph.D. thesis, Ecole Centrale Paris, CIFRE SDTools, 2011.
- [2] V. Jaumouillé, Dynamique Des Structures à Interfaces Non Linéaires : Extension Des Techniques de Balance Harmonique. PhD thesis, Ecole Centrale de Lyon, 2011.
- [3] C. Hammami, Intégration de Modèles de Jonctions Dissipatives Dans La Conception Vibratoire de Structures Amorties. PhD thesis, Arts et Metiers ParisTech, Paris, Oct. 14.
- [4] 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.