SDT non-linear transient and HBM toolbox 0.3

User guide

Initially funded by CLIMA a project supported by Conseil régional d'Ile-de-France and BPI

Etienne Balmes
Guillaume Vermot des Roches

SDTools
44 rue Vergniaud
75013 Paris (France)
Tel. 33 +1 44 24 63 71

Contents

1

s*theory

Theory and reference

Contents

s*nlconst

1.1 Non-linear system representation

_GenStrain

1.1.1 Strains, stresses, application of forces

One is interested in solving equations of the general form

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

eq*

with

- q the finite element DOFs (full or reduced)
- M,C,K the mass, viscous damping, and stiffness matrices respectively
- F_{ext} the external forces with the sign convention that they are written on the right hand side
- F_{NL} the non-linear forces internal to the system (sign convention on the left hand side of the equation) which may depend on the FEM motion described by its generalized strains and possibly strain rates defined below.
- When internal states are necessary (friction, plasticity, ...), they are stored either as generalized strain components or DOF and 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

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

eq*

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 1.2.1 , surfaces in section 1.2.2 , volumes in section 1.2.3 .

For *time domain*, generalized strains ϵ (noted `.unl` in the code) are obtained by computing

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

eq*

In a similar fashion, if the `NL.vnl` field exists strain rates are obtained using

$$v_{nl} = [c]\left\{q'\right\} + v_{nl0}. \tag{1.4}$$

eq*

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

$$\{u_{nl}\}_{(N_E\times N_G)\times N_T} = \{\epsilon\}_{\text{gauss repeat}\times\text{time}} = \begin{bmatrix} e_{1,g1}(t_1) & e_{1,g1}(t_2) & \dots \\ e_{2,g1}(t_1) & \dots & \\ \vdots & & \\ e_{1,g2}(t_1) & \dots & \\ \vdots & & \end{bmatrix} \tag{1.5}$$ eq*

- From the observed strains, a constitutive law is used to **estimate stresses** (or generalized forces), as will be discussed in section 1.3 from data present in the `NL.Fu` field,

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

  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

$$\{-F_{nl}\} = [b]\{s_{nl}\} \tag{1.7}$$ eq*

  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,

- 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 1.3.3 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.

ljacobian

1.1.2 Jacobian computations

Many iterative schemes need derivatives of the non-linear forces with respect to strains (stiffness)

$$k_j = [b]\left[\frac{\partial s_{nl}}{\partial u_{nl}}\right][c] \tag{1.8}$$

eq*

or strain rates (damping matrix)

$$c_j = [b]\left[\frac{\partial s_{nl}}{\partial v_{nl}}\right][c] \tag{1.9}$$

eq*

See `nl_spring` `NLJacobianUpdate` for low level documentation and `nl_solve` `TgtMdl` for global calls. The derivatives at each integration point (often named Gauss point in the documentation) $\frac{\partial s_{nl}}{\partial u_{nl}}$ correspond to the material stiffness matrix called `dd` or `Lambda` in different parts of SDT. Since the `b` and `c` matrices combine all gauss points, the assembled matrix containing one block for each gauss point, is sometimes stored in a `ddg` field when individually modifying the gauss points is of interest (topology optimization applications for example).
When splitting of Jacobians is of interest xxx.

s*nlkin

1.2 Kinematics of non-linear systems

s*nlio1d

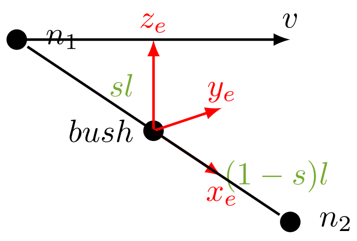
1.2.1 Generalized non-linear springs

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

$$dq = \left[-\{d\}^T \;\; \{d\}^T\right]\left\{\begin{matrix} u_1 \\ v_1 \\ w_1 \\ u_2 \\ v_2 \\ w_2 \end{matrix}\right\} \tag{1.10}$$

eq*

The non-linear implementation of the OpenFEM `cbush` element provides 6 generalized strains at a given location corresponding to relative translations and rotations. For `EDID==-1` this is with respect to a local basis $B = [x_e \; y_e \; z_e]$.



$$\left\{\begin{matrix} du \\ dv \\ dw \\ ru \\ rv \\ rw \end{matrix}\right\}_{NE\times 1} = \begin{bmatrix} -B^T & 0 & B^T & 0 \\ 0 & -B^T & 0 & B^T \end{bmatrix} \left\{\begin{matrix} u_1 \\ v_1 \\ w_1 \\ ru_1 \\ rv_1 \\ rw_1 \\ u_2 \\ v_2 \\ w_2 \\ ru_2 \\ rv_2 \\ rw_2 \end{matrix}\right\} \tag{1.11}$$

eq*

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 1.3.6
- `.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 .
When considering non small angular rotations of the two nodes, the orientation basis in (1.11) cannot be considered considered constant. It is thus necessary to express motion of each part at the bushing node using large rotations

$$\{x_{iB}(t)\} = \{x_i(t)\} + [R_i(t)]\{x_{iB}(0) - x_i(0)\} \tag{1.12}$$

eq*

In non-linearities that implement such large rotation, SDT expects that the bushing frame is associated with the first node and that motion of the bushing node associated with the second.

Thus the relative translation in first node frame is given by

$$[R_1(t)]^T \{x_{2B} - x_{1B}\} = [R_1(t)]^T \{x_2(t) - x_1(t) - x_1(0) + x_{1B}(0)\} + [R_1(t)]^T [R_2(t)] \{x_{2B}(0) - x_2(0)\} \quad (1.13)$$

eq*

For rotational springs at the bushing, there is no obvious large rotation formulation. The current implementation uses the difference of angles even though this does not always make sense.

s*nlio2d

1.2.2 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 are implemented in `p_zt`. When an `NLdata` field is defined a vectorized non-linearity is initialized.
Contact elements, implemented in `p_contact` have an objective similar to that of zero thickness element but allow the handling of non-conform meshes. Subtype 1 does not account for large displacement so that matching can be performed once and kept constant during time. Subtype 2 allows for relative motion of two bodies (wheel moving on a rail for example) and currently only works for matching to triangular surfaces.

s*nlio3d

1.2.3 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).
For volumes, the choice of *strain* is obtained using the `p_solid` `Isop` parameters. For explicit mechanics in large deformation, the displacement gradient is used and obtained using `Isop=100`.
For generic implementation of *multi-physic non-linear volumes*, a generalized strain giving the displacement and its gradient for each field. For a field with 3 components at N nodes

$$\{e_{MP}\}_{12} = \left\{\begin{matrix} u_1 \\ u_{1,j} \\ u_2 \\ u_{2,j} \\ \vdots \end{matrix}\right\}_{12} = \begin{bmatrix} N & 0 & 0 \\ N,x & 0 & 0 \\ N,y & 0 & 0 \\ N,z & 0 & 0 \\ 0 & N & 0 \\ 0 & N,x & 0 \\ 0 & N,y & 0 \\ 0 & N,z & 0 \\ 0 & 0 & N \\ 0 & 0 & N,x \\ 0 & 0 & N,y \\ 0 & 0 & N,z \end{bmatrix} \left\{\begin{matrix} u_1(n1) \\ u_2(n1) \\ u_3(n1) \\ \vdots \end{matrix}\right\}_{3N} \quad (1.14)$$

eq*

$$= [B_{MP}(r,s,t)]_{12\times 3N} \{u_{in}\}_{3N}$$

As a an example, the standard linearized mechanical strain

$$\left\{\begin{array}{c}\epsilon_x\\ \epsilon_y\\ \epsilon_z\\ \gamma_{yz}\\ \gamma_{zx}\\ \gamma_{xy}\end{array}\right\}=\left[\begin{array}{ccc}N,x&0&0\\0&N,y&0\\0&0&N,z\\0&N,z&N,y\\N,z&0&N,x\\N,y&N,x&0\end{array}\right]\left\{\begin{array}{c}u\\v\\w\end{array}\right\}=[B_{Ep}]_{6\times3N}\,\{u_{in}\}_{3N} \tag{1.15}$$

eq*

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

$$[B_{Ep}]_{6\times3N}=\left[\begin{array}{cccc cccc cccc}0&1&0&0&0&0&0&0&0&0&0&0\\0&0&0&0&0&0&1&0&0&0&0&0\\0&0&0&0&0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&1&0&0&1&0\\0&0&0&1&0&0&0&0&0&1&0&0\\0&0&1&0&0&1&0&0&0&0&0&0\end{array}\right][B_{MP}]_{12\times3N}=[T_{Ep}]\,[B_{MP}] \tag{1.16}$$

eq*

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

$$K_e=\int_\Omega [B_{Ep}]^T\,[D]\,[B_{Ep}]\,d\Omega=\int_\Omega [B_{MP}]^T\,\left[T_{Ep}^T D T_{Ep}\right]\,[B_{MP}]\,d\Omega \tag{1.17}$$

eq*

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 constitutive 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 $T_{Ep}^T D T_{Ep}$ in (1.17).

s*nlbcred

1.2.4 Kinematic reduction, observation and hyperreduction

Once a kinematic reduction defined, where $\{q\} = [T]\{q_R\}$ is defined, non-linear observation and command matrices are simply reduced using $c_R = c * T$ and $b_R = T^T b$.
For resultant observations, the strategy is however more complex. In a frequency domain model, the resultant is associated with $Z_I(p,s)$ the dynamic stiffness restricted to an interface

$$R_I == [c_I]\,[Z_I(p,s)]\,\{q(\omega)\} = [c_I]\left[Ms^2 + \sum \alpha_p M_p\right]\{q(\omega)\} \tag{1.18}$$

eq*

so that the coefficients of the observation depends on frequencies and parameters involved in $Z_I(p,s)$. In SDT, such linear combinations are described in `zcoef`. Taking the case of a bushing where $[Z_I] = E_(\omega,a)\,[K_u]$, the resultant observer is simply given by $[c_I]\,[Z_I] = E_(\omega,a)\,[c_I K_u]$.
When the observer is built around an operating condition where large displacement has occurred, the observation matrix may depend on the current position $c(x)$. For example, the elongation of a large rotation rod is given by $dl(t) = \sqrt{\{u_2-u_1\}^T\{u_2-u_1\}} - l_0$, so that the linearized observation is given by

$$dl(t) = \{e_i(u_0)\}^T\{u_{i2}-u_{i1}\} = \frac{1}{\|u_2(0)-u_1(0)\|}\{u_2(0)-u_2(0)\}^T\{u_{i2}-u_{i1}\} \tag{1.19}$$

For the case of resultant observation in enforced displacement problems, $R_I(t) = [T_I]^T F(t)$, the reduced equations of motion give an estimate of $F_R(t)$ and not F_T, thus T_{IR}^T should be used when observing.
Hyperreduction is a strategy where kinematic reduction is combined with selection of a subset of non-linear integration points based on criteria on the work of reduced model. Thus from a set E of Gauss points where each non-linear strain is described by $u_{nl}^g = [c^g]\{q\}$, one restricts to a subset E_{HR}. Needs more details.

s*nlbc

1.2.5 Manual definition of input and output (deprecated)

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).

s*nlfu

1.3 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.

s*nlfubase

1.3.1 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

$$\{s_{nl}(t)\} = F\left(u_{nl}, v_{nl}\right) \tag{1.20}$$

eq*

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 1.3.2 .
- dedicated user function section 1.3.6 (deprecated).
- anonymous function defined with parameters section 1.3.7 (deprecated).

*nltabular

1.3.2 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 (used in `nl_inout FuTable`), ie 1st column is the observed strain (relative displacement), 2nd column F_{u1} is the corresponding stress (load). A third column F_{u2} 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). Thus $s_{nl} = F_{u1}(u_{nl}) + F_{u2}(u_{nl})F_v(v_{nl})$.

  When each non-linearity has 3 strains, this is interpreted as contact. You are expected to have non-linear kinematics giving strains as normal component followed by two tangent directions. Thus $s_1 = F_{\text{normal}} = F_{u1}(u_1)$ and $s_2 = F_{\text{Tangent1}} = F_{u2}(u_1)F_v(v_2)$ $s_3 = F_{\text{Tangent2}} = F_{u2}(u_1)F_v(v_3)$.

  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, abscissa 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')`.

For Jacobian computations by `nl_spring NLJacobianUpdate`, one uses xxx
Supported data input forms

- `BumpStop dp` *dp* `kp` *kp* `cp` *cp* `dm` *dm* `km` *km* `cm` *cm* (a bumpstop example can be found in `sdtweb t_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.

- `Friction f` *f* `c0` *c0*, where *f* is the friction load, and *c0* is a damping coefficient applied on the transition around 0 velocity (*c0* is typically important). Dry frictions are known to be responsible for convergence problems.

- `gapCylI` inner cylinder within an external sleeve.

- `ctcFric` penalized contact and friction implementation

nstintern

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

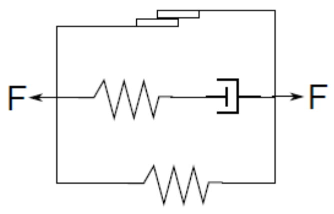


Figure 1.1: Elastic-visco-plastic model fig

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

$$f_{NL} = \mathcal{F}\left(\{q\},\{\dot{q}\},\{z\},\{\dot{z}\},t\right) \tag{1.21}$$

eq*

and the internal states evolution equation functional

$$\{\dot{z}\} = \mathcal{G}\left(\{q\},\{\dot{q}\},\{z\},t\right) \tag{1.22}$$

eq*

where $\{z\}$ is a vector of internal states, $\{q\}$ the displacement vector and $\dot{x}$ represent the time derivatives.
Equations (1.21) and (1.22) represent the class of so-called state space models (for example in the 80's for geophysics applications, in particular by Rice and Ruina rice_1983 [1]).
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 (1.21) and (1.22) do not comply with a second order based resolution framework (the internal states acceleration is seldom defined).
Equation (1.22) 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

$$\{u_{nl}\}_{((N_E+N_I)\times N_G)\times N_T} = \{\epsilon\}_{\text{gauss repeat}\times\text{time}} = \begin{bmatrix} e_{1,g1}(t_1) & e_{1,g1}(t_2) & \dots \\ e_{2,g1}(t_1) & \dots & \\ \vdots & & \\ e_{1,g2}(t_1) & \dots & \\ \vdots & & \\ e_{1,i1}(t_1) & & \\ \vdots & & \end{bmatrix} \quad (1.23)$$

eq*

This formalism keeps vectorization capability per instant. The internal rate states are stored in the same manner.
Strain history is stored in the t̂hird dimension `.unl(:,:,jh)`. In general for time integration, one will use

- `.unl(:,:,1)` to store the current strains of the form (1.3)
- `.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 (1.22) 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 (1.3) but assuming that internal states have not changed. The residual function computes a step replacing the data in `.unl(:,:,1)`. `StoreState` (done at end of `Fu`) copies `.snl` to `.FNL` . Copy of internal states to `unl(:,:,2)` can be done by controlled by the `Fu` (this is in particular is done in the `StoreState` C++ function when ûpUnl2=RO.jg[2]=2 is used).

- `StoreType=02` stores internal states in `.FNL` rather than `.snl`.

- `StoreType=03` same as 01 but `StoreState` stores the full `.snl` vector followed by the full `.unl(:,:,1)`. Beware that, this can be quite memory intensive.

- `StoreType=04` used for enforced motion. Modifies the displacement and velocity buffers.

- `StoreType=1..` if the hundreds digits is set to 1, `.vnl` is assumed not used and thus the associated buffer is emptied. xxx need discuss xxx

When `.iopt(1)=FInd` is positive, `.snl` is copied as a consecutive vector to `model.FNL` assuming the field exists. When `.iopt(2)=iu` is positive, internal states are copied consecutively to the displacement vector `u` with offset `iu`. 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 evolution, 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.

  `.99` adds the internal state as an additional DOF, while `-.99` uses an `FNL` internal state.
- 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 transformation, 9 components of stress gradient are observed. `.adofi=zeros(18,1)+.99` will interlace 18 internal states with the observation at each Gauss point.

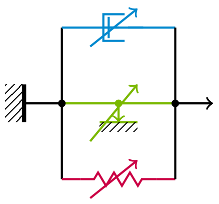
By default, one should let initialization procedures allocate DOF identifiers to each internal states by only providing a DOF extension in `.adofi`. If internal states have a physically defined nodal support, it is also possible to provide the corresponding DOF instead of just an extension. Beware that 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 (1.22), and then computes the non-linear forces with equation (1.21).
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 (1.21) is directly obtained. One then updates the internal states evolution equation (1.22) for convergence iterations. Internal states DOF must thus be kept in the resolution phase.

s*uMaxw

1.3.4 Hyper-visco-hysteretic 0D model

For the representation of bushings, each Gauss point is a scalar (0D) constitutive law that considers a combination of hyperelastic (red curve in figure 1.2), rate independent hysteresis (green curves from low speed triangular testing), and viscoelastic behavior (blue to yellow maps denoting frequency and amplitude dependence from sine testing).



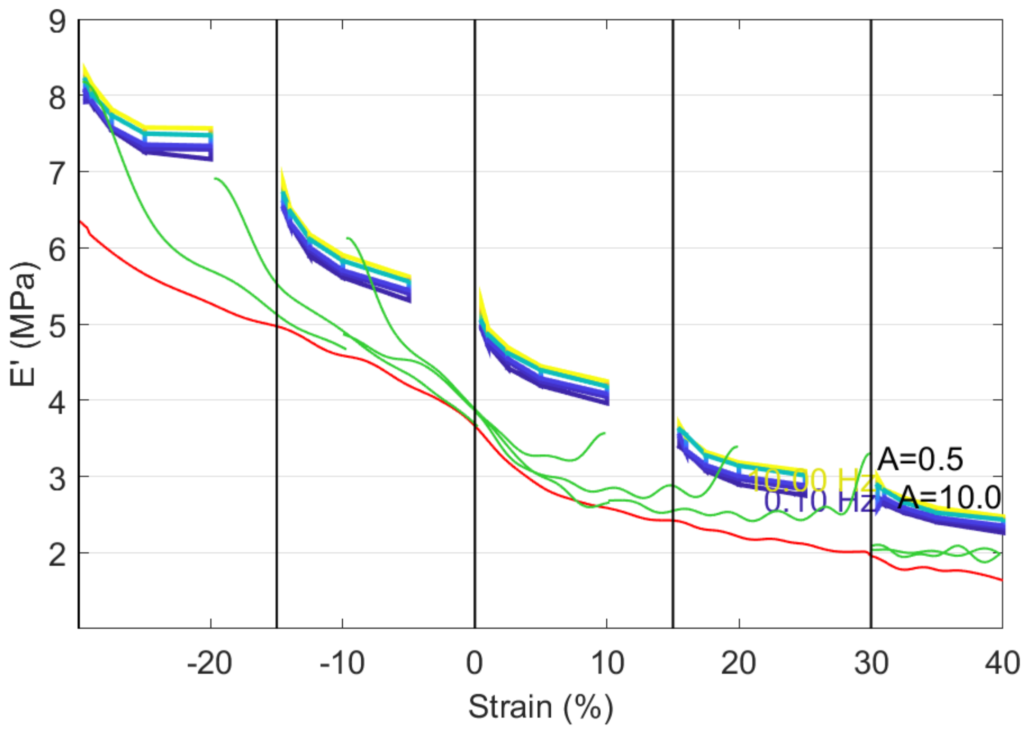


Figure 1.2: Tangent hyperelastic stiffness (red), hysteretic relaxation modulus (green), viscoelastic complex modulus (blue to yellow) f:0

The total load is a series of forces/stresses

$$F = \sum_{i=0}^{N_{cell}} F^i \tag{1.24}$$ eq*

where hyperelasticity is represented using a tabular form with either linear of piecewise cubic interpolation with no internal states $F^0_g(u_g)$ (stored as `NLdata.Fu{k}`)and other behavior is represented using a series of cells (also called branches in reference to the classical rheologic representation of figure 1.3), specified using a field `.cell` where each row describes a branch indexed by i with `[ty gi fi xfi ai]` type, load fraction, and if used frequency in Hz , sliding distance, non-linearity coefficient. xxx Note that the convention is to declare constants using a fraction of the high frequency modulus

$$g^0 + \sum_i g^i = 1 \tag{1.25}$$

but when starting relaxations from the low frequency stresses, the coefficient g^i/g^0 is more relevant and that coefficient only needs to be positive.

Currently supported types

- 4 stress rate relaxation (adds high frequency branches) with Lion transition between viscous and hysteretic behavior (for non-zero x_{fi}). A relaxation frequency $f_i = \omega^i/2\pi = K^i/(c^i 2\pi)$, a load fraction $g_i = K^i/K^\infty$, and a saturation distance x_{fi} (stored in a `NLdata.Fu{k}.cell` row containing `[8 gi fi(Hz) 1/xfi(1/disp,strain)]`) are used for the differential evolution law

$$\dot{F}^i + F^i\left(\omega_i + \frac{|\dot{u}|}{x_f^i}\right) = -\frac{g^i}{g^0}\dot{F}^0 \tag{1.26}$$ eq*

For the linear case, the following illustrates the contributions of branches in the frequency domain.

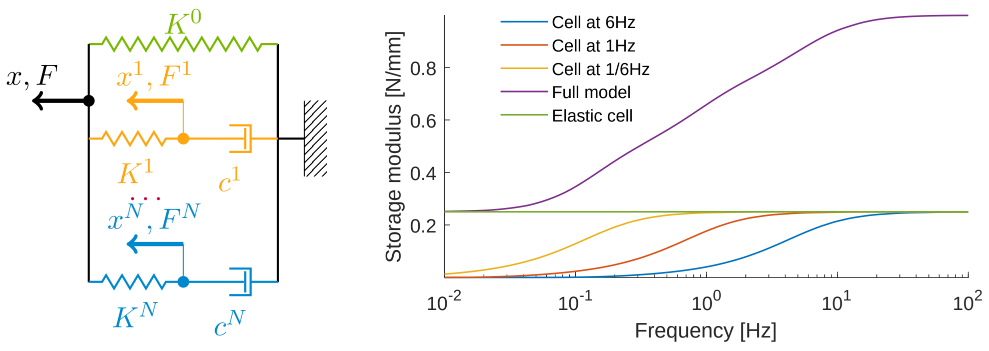


Figure 1.3: Generalized Maxwell model and associated frequency domain response. Stress/strain rate relaxation model adds $s/(s+\omega_j)$ with non-zero high frequency contribution (orange to blue contributions in the plot). f:m

An implicit approximation of the differential equation given by

$$\frac{F^i(t_{n+1}) - F^i(t_n)}{dt} + \left(\omega_i + \frac{|\dot{u}|}{x_f^i}\right) F^i(t_{n+1}) = \frac{g^i}{g^0}\,\frac{F^0(t_{n+1}) - F^0(t_n)}{dt} \tag{1.27}$$ eq*

leading to fixed time step increment equation

$$F^i(t_{n+1}) = \frac{F^i(t_n) + \frac{g^i}{g^0}\left(F^0(t_{n+1}) - F^0(t_n)\right)}{1 + dt\left(\omega_i + \frac{|\dot{u}|}{x_f^i}\right)} \tag{1.28}$$ eq*

Alternatively an explicit approximation in F^i with $F^0(t_{n+1})$ known

$$\frac{F^i(t_{n+1}) - F^i(t_n)}{dt} + \left(\omega_i + \frac{|\dot{u}(t_n)|}{x_f^i}\right) F^i(t_n) = \frac{g^i}{g^0}\,\frac{F^0(t_{n+1}) - F^0(t_n)}{dt} \tag{1.29}$$ eq*

leading to fixed time step increment equation

$$F^i(t_{n+1}) = F^i(t_n) - dt\left(\omega_i + \frac{|\dot u(t_n)|}{x_f^i}\right)F^i(t_n) + \frac{g^i}{g^0}\left(F^0(t_{n+1}) - F^0(t_n)\right) \tag{1.30}$$ eq*

xxx recheck

- 1 viscoelastic relaxation of strain removes low frequency branches $-g^i/(s/\omega_j+1)$ from the total response. Using one internal state u^i associated with each Gauss point, a relaxation frequency $f_i = \omega^i/2\pi = K^i/(c^i 2\pi)$ and a load fraction $g_i = K^i/K^\infty$ (stored in a `NLdata.Fu{k}.cell` row containing `[1 gi fi(Hz) 0]`), the evolution is described by

$$\frac{\dot u^i}{\omega_i} = (u - u^i) \text{ and } F^i = (K^\infty g^i)(u-u^i) = (K^\infty g^i)\left(1 - \frac{1}{s/\omega^i + 1}\right)u \tag{1.31}$$ eq*

Note that F^0 is here the high frequency asymptote. The implicit increment equation uses $u^i(t_{n+1})(1+\omega_i dt) = \omega_i dt\ u(t_{n+1}) + u^i(t_n)$ leading to

$$u^i(t_{n+1}) = \left(\omega_i dt\ u(t_{n+1}) + u^i(t_n)\right)\frac{1}{(1+\omega_i dt)} \tag{1.32}$$ eq*

- 2 viscoelastic relaxation of strain rate (stress rate is better), internal state u^i

$$\dot u^i + \omega_i u^i = \dot u \text{ and } F^i = (K^\infty g^i)u^i = (K^\infty g^i)\frac{su}{s+\omega^i} \tag{1.33}$$ eq*

- 3 viscoelastic relaxation of hyperelastic stress, internal state F^i

$$\frac{\dot F^i}{\omega_i} + F^i = -\frac{g^i}{g^0}F^0(u) \tag{1.34}$$ eq*

- 4 viscoelastic relaxation of hyperelastic stress rate, internal state F^i,

$$\dot F^i + \omega_i F^i = \frac{g^i}{g^0}\dot F^0(u) = \frac{g^i}{g^0}\left.\frac{\partial F^0}{\partial u}\right|_u \dot u \tag{1.35}$$ eq*

- 9 Lion transition between viscous and hysteretic behavior *for 3D deviatoric stress* $\|\dot u\|$ is a scalar per material point and not by stress component.

$$\dot F^i + F^i\left(\omega_i(1+\beta_i\|\dot u\|)\right) = \frac{g^i}{g^0}\dot F^0 \tag{1.36}$$

- `301` For 3D hyperelastic behavior without large deformation, one assumes a reference linear elastic behavior $\sigma = D\varepsilon$. The hyperelastic stress is assumed to be a function of $I(\varepsilon) = \varepsilon_{11} + \varepsilon_{22} + \varepsilon_{33}$ (used as an approximation of $\bar{I}_1 - 3$ see xxx) of the form

$$\sigma_0 = f(I(\varepsilon))\,[D]\,\{\varepsilon\} \tag{1.37}$$

the viscoelastic forces and then obtained using xxx

$$\dot{\sigma}^i + \sigma^i\left(\omega_i(1+\beta_i\,\|\dot{u}\|)\right) = \frac{g^i}{g^0}\dot{\sigma}^0 \tag{1.38}$$

and the non-linear strain contribution (where the linear contribution is removed if `keepLin>0`) xxx

$$s_{nl} = (g_0 - 1)\,[D]\,\{\varepsilon\} + \sum_i \sigma^i \tag{1.39}$$

- `5` friction element (Iwan model, Jenkins cell) with simple target, parameters are sliding distance $x_f^i = F_f^i/K^i$ and load fraction $g_i = K^i/K^\infty$. The load saturation formulation with internal state corresponding to the hysteretic force F^i and evolution equation

$$\begin{array}{ll} \dot{F}^i = \dfrac{g^i}{g^0}\dot{F}^0,\ \text{if } F^i\text{sign}(\dot{u}) < F_f^i = g_f^i F^0(u) = xxxwasK^i x_f^i & \text{sticking state} \\ \dot{F}^i = 0,\ \text{if } F^i\text{sign}(\dot{u}) = F_f^i & \text{sliding state} \end{array} \tag{1.40}$$ eq*

is preferred to the classical displacement formulation with internal state u^i position of last turning point,

$$\begin{array}{ll} \dot{u}^i = 0,\ \text{if } \left\|u - u^i\right\| < x_f^i = \frac{F_f^i}{K^i} & \text{sticking state} \\ \dot{u}^i = \dot{u},\ \text{if } \left\|u - u^i\right\| = x_f^i = \frac{F_f^i}{K^i} & \text{sliding state} \end{array} \tag{1.41}$$ eq*

- `6` friction element with Dahl transition using internal state F^i, parameters $F_f^i = K^\infty g_i x_f^i$ limit force and speed exponent α^i (stored in `a_i` coefficient) associated with evolution equation

$$\dot{F}^i = \left(1 - \frac{F^i}{(g_f^i/g^0)F^0}\text{sign}(\dot{u})\right)^{\alpha_i} \frac{g^i}{g^0}\dot{F}^0, \tag{1.42}$$ eq*

- `7` friction element with Dahl stress relaxation using internal state F^i, parameters $F_f^i = K^\infty g_i x_T^i$ limit force and speed exponent α^i (stored in `NLdata.Fu{k}.g`, `.xf` and `.aDahl`) xxx associated with evolution equation

$$\dot{F}^i = (K^\infty g_i)\left(1 - \frac{F^i}{F_f^i}\text{sign}(\dot{u})\right)^{\alpha_i} \dot{u}, \tag{1.43}$$ eq*

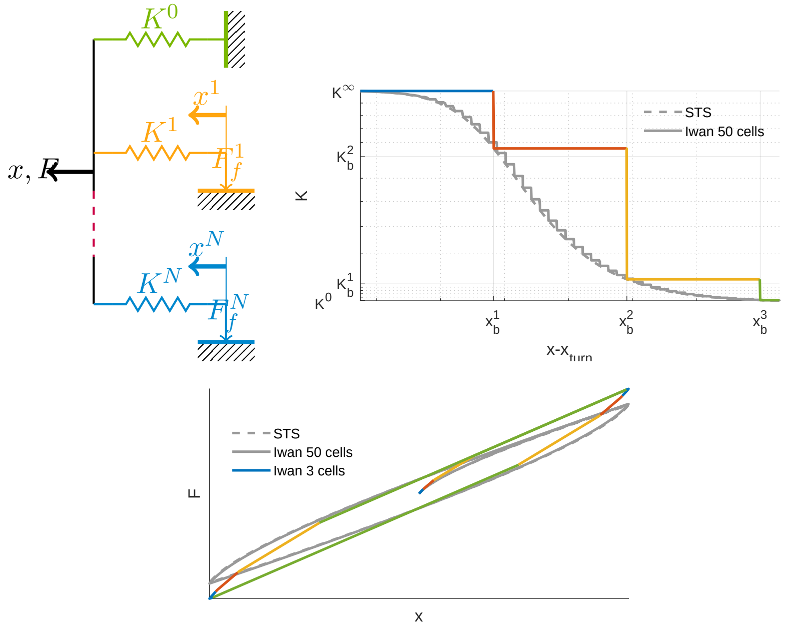


Figure 1.4: Scheme for Iwan model and respective response in terms of hysteretic relaxation and on force/displacement domain.

f:j

For runtime integration, the data necessary for evolution equations is stored as

- `.opt=[ FuCode=uMax tc dt0 ]` followed by a series of cell constants starting at opt(10).
- `.iopt` starts with the standard `Find, iu, Nstrain, Nistate, Ngauss(5)`, then a list of operations are specified starting from `tcell(10)` and `cval=opt(10)`. Implemented values are listed using `sdtm.enum('nlUmaxw')`
  - `tcell=-1,cval=[]` last cell
  - `tcell=1,cval=[v1 v2 v3]` viscoelastic (Maxwell) relaxation of strain, 3 `cval` values give the recursion $u^i := (u^i v_2 + u^0)/v_1$ and load fraction definition $F^i = v_3 u^i = K^i u^i$. $v_1 = \omega_i$ in rad/s, $v_2 = g_i/g_0$, if non-zero $v_3 = 1/x_{fi}$.
  - `2, cval=[v1 v2 v3]` viscoelastic relaxation of strain, values in `cval` give the recursion $u^i := (u^i v_2 + v^0)/v_1$ and gain $F^i = v_3 u^i = K^i u^i$
  - `3, cval=[v1 v2 v3]` viscoelastic relaxation of stress, values in `cval` give the recursion $F^i := (F^i v_2 + v_3 F^0)/v_1$.
  - `4, cval=[v1 v2 v3]` viscoelastic relaxation of stress rate, values in `cval` give the recursion $F^i := (F^i v_2 + v_3 \dot{F}^0)/v_1$. xxxLion saturation needs documenting

- Slip1D,Slip2D friction using 1 or 2 sliding directions use 4, cval=[Mu KtLin kappa dMudV Sat KtSat]]
- 101 F0_dim1 F0_dim2 interpolation table to estimate a scalar $s_{nl}(u_{nl})$. Supported values for dim2 are listed in sdtm.enum('nlFu') (linear, piecewise cubic, analytic forms, ...)
- 102 linear scalar stiffness, one constant
- 103 linear scalar viscosity
- tcell=[1000 unlshift snlshift] component change.
- tcell=300, cval=[c1, c2, c3, kappa, kappav] hyperelastic material cell he MooneyRivl CiarletGeymonat
- tcell=301, cval=[omegai,gi/g0,ef] relaxation of deviatoric part of S given by $\dot{S}^i + (\omega^i + \frac{\|d\|}{\epsilon_f})S^i = \frac{g^i}{g^0}\dot{S}^0$ xxx discuss (2.39) with Rafael xxx
- tcell=[400] xxx non usual interlacing Ng for first strain g, Ng of u_{t1}, u_{t2},

The constants v_i depend on the integration scheme and constitutive laws

- For implicit integration of Maxwell form (1.31), the evolution equation of the internal state is

$$u^i(t_{n+1})\left(\frac{1}{\omega^i dt}+1\right) = \frac{u^i(t_n)}{\omega^i dt} + u(t_{n+1}) = u^i(t_{n+1})v_1 = u^i(t_n)v_2 + u(t_{n+1}) \tag{1.44}$$ eq*

- For implicit integration of strain rate (1.33), the evolution equation of the internal state is

$$u^i(t_{n+1})\left(\frac{1}{dt}+\omega^i\right) = \frac{u^i(t_n)}{\delta t} + \dot{u}(t_{n+1}) = u^i(t_{n+1})v_1 = u^i(t_n)v_2 + \dot{u}(t_{n+1}) \tag{1.45}$$ eq*

- For implicit integration of stress and stress rate, simply replace u with F.

s*MexIOa

1.3.5 User callback in nl_inout, MexCb field

The current high performance developments are focusing on vectorized user implementation of non-linearities integrated in the chandle nl_inout implementation. In the data structure used during time integration (see section 1.6.3), the .MexCb field is used to provide data. It is a cell array giving {@fun,RO}.
For optimized operation limiting field name checks, the option structure RO is assumed to have ordered fields

- .unlg at a single Gauss point,
- .opt copy of NL.opt,

- `.jg` int32 vector allowing passing of current Gauss point index,
- `.vnlg` optional copy of velocities at Gauss point or empty.
- `.mID` is initialized to the chandle/mhandle C pointer when needing a callback to modify non-linearity data.

Expected return arguments are `snl` and `uint` (new value of internal states).

s*nluser

1.3.6 Dedicated user function (deprecated)

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

```
function NL=resCubic(NL,fc,model,u,v,a,opt,Case,RO)
NL.unl=-.01*NL.unl.^3  ... % Cubic stiffness on relative motion
     + -.02*NL.vnl;        % Linear viscous damping on relative velocity
```

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

.anonymous

1.3.7 .anonymous field for definition (deprecated)

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 `@(...)` 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 string 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.

```
d_hbm('TestDuffing2Dof-an')
% completes the defintion
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);
```

maxwellex

1.3.8 Maxwell cell model using matrices (deprecated)

The Maxwell cell 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 1.5 illustrates the Maxwell (or Zener) model.

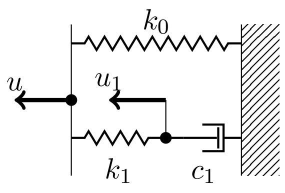


Figure 1.5: Standard viscoelastic model (single cell Maxwell or Zener). Generalized multi-cell model.

axwellCell

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

$$\dot{e}_n = \frac{K_n}{C_n}(e_n - e_b) \tag{1.46}$$

eq*

and recompose the total non-linear stress using

$$s_{nlb} = K_0 e_b + \sum_n K_n(e_b - e_n) \tag{1.47}$$

eq*

To allow more general superelement 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.

$$\begin{bmatrix} K_{bb} & \mathbf{K}_{bi} \\ \mathbf{K}_{ib} & \mathbf{K}_{ii} \end{bmatrix} \begin{Bmatrix} e_b \\ \mathbf{e}_i \end{Bmatrix} + \begin{bmatrix} C_{bb} & \mathbf{C}_{bi} \\ \mathbf{C}_{ib} & \mathbf{C}_{\mathbf{ii}} \end{bmatrix} \begin{Bmatrix} \dot{e}_b \\ \dot{\mathbf{e}}_i \end{Bmatrix} = \begin{Bmatrix} s_{nlb} \\ \mathbf{0} \end{Bmatrix} \tag{1.48}$$

eq*

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

$$\dot{\mathbf{e}}_i = -\mathbf{C}_{ii}^{-1}\mathbf{K}_{ib}e_b - \mathbf{C}_{ii}^{-1}\mathbf{C}_{ib}\dot{e}_b - \mathbf{C}_{ii}^{-1}\mathbf{K}_{ii}\mathbf{e}_i \tag{1.49}$$

eq*

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

$$s_{nlb} = \begin{bmatrix} K_{bb} & \mathbf{K}_{bi} \end{bmatrix} \begin{Bmatrix} e_b \\ \mathbf{e}_i \end{Bmatrix} + \begin{bmatrix} C_{bb} & \mathbf{C}_{bi} \end{bmatrix} \begin{Bmatrix} \dot{e}_b \\ \dot{\mathbf{e}}_i \end{Bmatrix} \tag{1.50}$$

eq*

Resolution of equation (1.49) can be obtained with different strategies. In transient simulations a local integration is performed using the Euler scheme. From instant t_n the internal state at instant t_{n+1} has to be integrated over $h = t_{n+1} - t_n$

$$\mathbf{e}_i^{n+1} = \mathbf{e}_i^n + h(1-\theta)\dot{\mathbf{e}}_i^n + h\theta\dot{\mathbf{e}}_i^{n+1} \tag{1.51}$$

eq*

Implemented strategies can involve either single or multiple step integration in implicit or explicit form. Implicit resolution involves 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 (1.49) and (1.50) 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.

s*hyper3d

1.4 Hyper-visco 3D model

The SDT/OpenFEM implementation of volumes also partially supports hyper viscoelastic laws under large deformation. This section derived from [?]. This first section summarizes classical results of continuum mechanics, which are needed for later discussion of constitutive laws exhibiting hyper-elasticity, viscoelasticity and rate independent hysteresis under large deformation [2, 3, 4].

salencon83,doghri00,sim

hyper3dkin

1.4.1 Kinematics

Considering a three dimensional euclidean space with base e_1, e_2, e_3, a body Ω may be described by a region from this space. Each point P of Ω is called a material point and its coordinates are described by

$$\{x(t)\} = x_1 \{e_1\} + x_2 \{e_2\} + x_3 \{e_3\} \tag{1.52}$$

in the reference configuration of a Lagrangian description of the problem. The body position may evolve with time, and time derivatives give velocity v and acceleration a as

$$v = \frac{\partial x}{\partial t}, \quad a = \frac{\partial^2 x}{\partial t^2}. \tag{1.53}$$

The kinematic description is based on the knowledge of the initial and current positions of any material point. The difference between initial and current positions of material points is the displacement vector $\{u\}$. Material behavior is typically described as a function of strains. SDT uses the simplest strain measure, the displacement gradient, given by

$$F = \frac{\partial \{x\}}{\partial \{x_0\}} = \delta_{ij} + \frac{\partial u_i}{\partial x_j} = \delta_{ij} + u_{i,j} \tag{1.54}$$

where δ_{ij} denotes the Kronecker delta function. This tensor links the reference and the current configurations, making it a hybrid deformation description. It may be divided in two separate parts: rotation R and dilatation U in a polar decomposition

$$F = RU \tag{1.55}$$

where R is orthonormal, and U positive definite and symmetric. To obtain rotation independent deformation tensors, quadratic forms of the deformation tensor are used. For a Lagrangian description, the right Cauchy-Green tensor

$$C = F^T F \tag{1.56}$$

will be used (the left Cauchy-Green tensor is $B = FF^T$). The Green Lagrange strain $e_{ij} = \frac{1}{2}(C_{ij} - \delta_{ij})$ is a common alternative. The main difference between the two is that C tends to the identity for nil deformation, while e tends to zero.

dinvariant

1.4.2 Invariants, strain rates

For isotropic materials, properties should not depend of orientation and behavior should not dependent on rotation R. Invariants of the strain tensors are often used in constitutive formulation to achieve rotation independence. The three classical invariants are

$$\{I_i\} = \left\{ \begin{array}{c} \mathrm{tr(C)} \\ \dfrac{\mathrm{tr}^2(C) - \mathrm{tr}(C^2)}{2} \\ \det(C) \end{array} \right\} \tag{1.57}$$

where one should note that $I_3 = \det(C) = \det^2(F)$ quantifies the square of the volume change at the material point, and $J = I_3^{1/2}$ the volume change. The derivatives of invariants with respect to deformations will also be necessary and are classically given by

$$\frac{\partial I_k}{\partial C_{ij}} = \begin{bmatrix} \delta_{ij} & I_1\delta_{ij} - C_{ij} & I_3 C_{ij}^{-1} \end{bmatrix}, \tag{1.58}$$

and

$$\frac{\partial I_k}{\partial C_{ij}\partial C_{kl}} = \left[0_{ijkl} \quad \delta_{ij}\delta_{kl} - 1_{ijkl} \quad I_3\left(C_{ij}^{-1}C_{kl}^{-1} - \tfrac{1}{2}\left(C_{ik}^{-1}C_{jl}^{-1} + C_{il}^{-1}C_{jk}^{-1} \right)\right)\right] \tag{1.59}$$

where $1_{ijkl} = \frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$. For highly incompressible materials, it is usual to describe the potential associated with the deviatoric part of the strain using reduced invariants, defined by

$$\bar{I}_1 = J^{-2/3} I_1 = I_3^{-1/3} I_1, \bar{I}_2 = J^{-4/3} I_2 = I_3^{-2/3} I_2. \tag{1.60}$$

It is useful to remind that for zero deformation $F = I$, $I_i = \begin{Bmatrix} 3 & 3 & 1 \end{Bmatrix}$ so that it makes sense to use expressions involving $\bar{I}_1 - 3$ (as in the Yeoh model). To check values on also reminds that $dI1dC = I, dI2dC = 2I, dI3dc = I$

$$D^D = 4 \begin{bmatrix} 1 & 1 & 1 & & & \\ 1 & 1 & 1 & & & \\ 1 & 1 & 1 & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix} \left(\frac{\partial^2\psi^I}{\partial I_3^2} + 2\frac{\partial^2\psi^I}{\partial I_2 I_3}\right) + 4 \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/2 \end{bmatrix} \left(\frac{\partial\psi}{\partial I_2} + \frac{\partial\psi}{\partial I_3}\right) \tag{1.61}$$

xxx
Deformation rate tensors are also often needed. The velocity gradient tensor is commonly used and given by

$$L = \frac{\partial v_i}{\partial u_j} = \frac{\partial v_i}{\partial x_j}\frac{\partial x_i}{\partial u_j} = \dot{F}F^{-1} \tag{1.62}$$

As this gradient is not rotation independent, it is handy to decompose it into strain rate and spin tensors, by taking its symmetric and the skew-symmetric parts,

$$D = \frac{1}{2}\left(L + L^T\right), W = \frac{1}{2}\left(L - L^T\right). \tag{1.63}$$

The strain rate tensor may be used in viscoelastic laws while the spin tensor is usually discussed in finite deformation plasticity models simo84 [5].

er3dstress

1.4.3 Stress, equations of motion, time integration at a material point

When computing work of deformable bodies the dual quantities (energy conjugate pair) are stress and strain, as force and displacement are dual for rigid body mechanics.
In the Lagrangian reference frame, the second Piola-Kirchhoff $S = JF^{-1}\sigma F^{-T}$ tensor is used for stress (degenerates to stresses in the original configuration for small deformation but large rotation). The hybrid reference tensor (initial frame for strains to current frame for stresses) is called first Piola-Kirchhoff $\Pi = J\sigma F^{-T}$ (or Boussinesq, or also nominal stress) and is not symmetric. In linearized conditions, the Eulerian description given by the Cauchy stress σ is used.
Stress and strain energy present different equivalent expressions, forming dual pairs

$$W = S : e = \Pi : F = \sigma : \varepsilon. \tag{1.64}$$

Material or constitutive models are laws that describe the evolution of stresses for an history of deformations and possibly internal states. They are system models of what happens at a material point.
Given a displacement gradient tensor, stress at timestep $n+1$ must be an explicit function of the gradient displacement tensor and its internal states at the last timestep n (considering explicit implementation),

$$[S_{n+1}, u_{n+1}^{int}] = f(\nabla u_{n+1}, \nabla \dot{u}_{n+1}, u_n^{int}). \tag{1.65}$$ eq*

The development of material laws described by (1.65) should typically be tested and tuned separately from the global structural application.

With material laws defined, equations of motion at the structure level are derived from the fact that the work of all forces for any kinematically acceptable virtual displacement field w is equal to zero. In other words,

$$\int_{\Omega_0} S : \delta e(w) - \int_{\Omega_0} \{f_v\}\,\delta w = 0. \tag{1.66}$$ eq*

Implicit solution of this equation typically require a tangent stiffness obtained from integration of the tangent material stiffness (1.69).

3d_he_int

1.4.4 Hyperelastic potential, polyconvexity

A material is said to be hyperelastic, if its stress only depends on the current strain. It is classically shown that this is verified if stress is obtained as the derivative of a potential ψ that depends on deformation tensors. Derivation from a potential is a sufficient condition to demonstrate that deformations are completely reversible, despite being nonlinear. The non-linearities involved are

- geometric: large deformations is typically involved requiring non-linear strain descriptions
- material: as the operating range can be large, there is no physical reason for materials to respond linearly

For an isotropic material, behavior must be independent of rotation. This can be used to demonstrate that the potential may always be expressed as a function of strain invariants. The stress is thus found as

$$S_{ij} = \frac{\partial \psi}{\partial e_{ij}} = 2\frac{\partial \psi}{\partial C_{ij}} = 2\frac{\partial I_k}{\partial C_{ij}}\frac{\partial \psi}{\partial I_k} \tag{1.67}$$ eq*

where the convention of summing repeated indexes is used. Note that the potential is decomposed in deviatoric (section 1.4.6) and compression (section 1.4.5) behavior so that the vector actually used is `dWdI`$=\left\{\partial\psi^D/\partial I_1 \quad \partial\psi^D/\partial I_2 \quad \partial\psi^D/\partial I_3 \quad \partial\psi^I/\partial I_3\right\}$.
Other consequence of the expression of the potential in terms of the invariants is that it may always be written in the form doghri00 [3]

$$\sigma = \beta_0(I_1,I_2,I_3)1 + \beta_1(I_1,I_2,I_3)F^TF + \beta_2(I_1,I_2,I_3)(F^TF)^2 \tag{1.68}$$ eq*

with β_i functions of the invariants that may be determined according to the model. These three β functions, therefore can be seen as a non parametric manner of modeling hyperelasticity. Since going through the whole space spanned by I_1, I_2 and I_3 is unpractical, most common approaches are to use either selected order representations as polynomials on the invariants or order independent models that are not based on invariants, as will be described in section 1.4.6 .

The only restriction on the energy potential is its polyconvexity salencon83,schroder03 [2, 6] which ensures stability. For scalar stress or 0D, the potential is the integral of the force/displacement curve. This function can be a polynomial or any convex function corresponding to the fact that the instantaneous stiffness or slope of the force/displacement curve is positive. In 3D, ensuring polyconvexity for a tensor is not straightforward but corresponds to the positive definite nature of the material stiffness. For an isotropic material, this material stiffness can be expressed as inria_2004,R5.03.19 [7, 8]

$$\mathcal{D}_{ijkl} = 4\frac{\partial^2 \psi}{\partial C_{ij}\partial C_{kl}} = 4\frac{\partial^2 I_n}{\partial C_{ij}\partial C_{kl}}\frac{\partial \psi}{\partial I_n} + 4\frac{\partial I_n}{\partial C_{ij}}\frac{\partial^2 \psi}{\partial I_n \partial I_m}\frac{\partial I_m}{\partial C_{kl}} \tag{1.69}$$ eq*

leading to the expression of the structural geometric stiffness bal2 [9]

$$[K_G] = \int_\Omega \{\delta q_m\}\{N_{,l}\}\left(F_{mk}\mathcal{D}_{ijkl}F_{ni} + S_{lj}\right)\{N_{,j}\}\{dq_n\} \tag{1.70}$$

eq*

SDT uses naming conventions dWdI,d2WdI2 for $\frac{\partial\psi}{\partial I_n}, \frac{\partial^2\psi}{\partial I_n \partial I_m}$ and expressions of these values for classic materials are given in section 1.4.5 for compression or isotropic volume change and section 1.4.6 for deviatoric or shear behavior.

yper3diso

1.4.5 Compression or volume change behavior

Materials used for bushings are typically incompressible. Compressibility is described by the isotropic part of strains and a specific discussion of constitutive laws is needed. Remaining behavior is associated with deviatoric strains and discussed separately.
To achieve this separation for hyperelastic models, one may divide the potential in a deviatoric and an isotropic part $\psi = \psi^D + \psi^I$, which means that the same decomposition applies to the stresses, aside from numerical issues. This implies the introduction of two separate models that are superposed in the end.
As the third invariant is the only one that has no deviatoric components, compression models are normally based on $J = I_3^{\frac{1}{2}}$. The isotropic potential for linear models normally used is

$$\begin{aligned}
\psi^I &= \frac{\kappa}{2}(J-1)^2 = \frac{\kappa}{2}(I_3 - 2I_3^{\frac{1}{2}} + 1) \\
\frac{\partial\psi^I}{\partial J} &= \kappa(J-1) = -p \\
\frac{\partial\psi^I}{\partial I_3} &= \kappa(1 - I_3^{-1/2}) \\
\frac{\partial^2\psi^I}{\partial I_3^2} &= \tfrac{\kappa}{2} I_3^{-3/2}
\end{aligned} \tag{1.71}$$

eq*

This potential however allows full compression or volume suppression. For this reason, the Ciarlet-Geymonat chapelle10 [10] potential was developed with a logarithmic penalty term,

$$\begin{aligned}
\psi^I &= \kappa\left(J - 1 - \ln(J)\right) = \kappa\left(I_3^{1/2} - 1 - \ln(I_3^{1/2})\right) \\
\frac{\partial\psi^I}{\partial J} &= \kappa\left(1 - \frac{1}{J}\right) = -p, \quad \frac{\partial^2\psi^I}{\partial J^2} = \kappa\left(1 + \frac{1}{J^2}\right) \\
\frac{\partial\psi^I}{\partial I_3} &= \frac{\kappa}{2}\left(I_3^{-1/2} - I_3^{-1}\right) \quad \frac{\partial^2\psi^I}{\partial I_3^2} = \kappa\left(-I_3^{-3/2}/4 + I_3^{-2}/2\right)
\end{aligned} \tag{1.72}$$

eq*

The stress tensor resulting from the isotropic part may be computed as $\sigma_p = p\delta_{ij}$ in an Eulerian reference frame, which may be transferred to a Lagrangian reference frame by $S_p = F^{-1}(\sigma_p)F^{-T}$.

per3dshear

1.4.6 Deviatoric laws

Three main groups of hyperelastic potentials are typically marckmann06 [11] distinguished: invariant, principal strains and physic based potentials.
Invariant based formulations were developed first and are still widely used, because of their simpler formulation and low numerical cost.
The **Yeoh** model uses polynomials of $(\bar{I}_1-3)=(J^{-2/3}I_1-3)=(I_3^{-1/3}I_1-3)$, $(\bar{I}_2-3)=(J^{-2/3}I_2-3)=(I_3^{-1/3}I_2-3)$, and $J-1=(I_3^{1/2})$ whose values vanish for zero displacement. And is given by

$$\begin{array}{rcl}
\psi^D &=& \sum_{i=1}^n C_{i0}(I_1 I_3^{-\frac{1}{3}}-3)^i \\
\dfrac{\partial\psi^D}{\partial I_i} &=& \left(\sum_{i=1}^n iC_{i0}(I_1I_3^{-\frac{1}{3}}-3)^{(i-1)}\right)\left\{I_3^{-\frac{1}{3}} \quad 0 \quad -\frac{1}{3}I_1I_3^{-\frac{4}{3}}\right\} \\
\dfrac{\partial^2\psi^D}{\partial I_i\partial I_j} &=& \left(\sum_{i=2}^n i(i-1)C_{i0}(I_1I_3^{-\frac{1}{3}}-3)^{(i-2)}\right)\begin{bmatrix} I_3^{-\frac{2}{3}} & 0 & -\frac{1}{3}I_1I_3^{-\frac{5}{3}} \\ 0 & 0 & 0 \\ -\frac{1}{3}I_1I_3^{-\frac{5}{3}} & 0 & \frac{1}{9}I_1^2I_3^{-\frac{8}{3}}\end{bmatrix} \\
&& +\left(\sum_{i=1}^n iC_{i0}(I_1I_3^{-\frac{1}{3}}-3)^{(i-1)}\right)\begin{bmatrix} 0 & 0 & -\frac{1}{3}I_3^{-\frac{4}{3}} \\ 0 & 0 & 0 \\ -\frac{1}{3}I_3^{-\frac{4}{3}} & 0 & \frac{4}{9}I_1I_3^{-\frac{7}{3}}\end{bmatrix}
\end{array} \quad (1.73)$$

used in (1.67) for stress computations and (1.69) for Jacobian computations. Note that Abaqus uses a generalization combining polynomials of the form $\sum_{i,j=1}^N C_{ij}(\bar{I}_1-3)^i(\bar{I}_2-3)^j$.
The **neo-hookean** model only considers C_{10} (asymptotic elasticity gives $G=2C_{10}$). The **Mooney-Rivlin** model adds the second invariant and constant C_{01} to this formulation and is given by

$$\begin{array}{rcl}
\psi^D &=& C_{10}\bar{I}_1+C_{01}\bar{I}_2 = C_{10}(I_1I_3^{-1/3})+C_{01}(I_2I_3^{-2/3}) \\
\dfrac{\partial\psi^D}{\partial I_i} &=& \left\{C_{10}I_3^{-\frac{1}{3}} \quad C_{01}I_3^{-\frac{2}{3}} \quad -\dfrac{C_{10}I_1I_3^{-\frac{4}{3}}}{3}-\dfrac{2C_{01}I_2I_3^{-\frac{5}{3}}}{3}\right\} \\
\dfrac{\partial^2\psi^D}{\partial I_i\partial I_j} &=& \begin{bmatrix} 0 & 0 & -\dfrac{C_{10}I_3^{-\frac{4}{3}}}{3} \\ 0 & 0 & -\dfrac{2C_{01}I_3^{-\frac{5}{3}}}{3} \\ -\dfrac{C_{10}I_3^{-\frac{4}{3}}}{3} & -\dfrac{2C_{01}I_3^{-\frac{5}{3}}}{3} & \dfrac{4C_{10}I_1I_3^{-\frac{7}{3}}}{9}+\dfrac{10C_{01}I_2I_3^{-\frac{8}{3}}}{9}\end{bmatrix}
\end{array} \quad (1.74)$$

eq*

where asymptotic elasticity gives $G=2(C_{10}+C_{01})$.
Stability requires $\partial^2\psi^D/\partial C^2$ to be a positive definite matrix, but the above potentials do not verify stability for some strain. A very simple potential featuring a very good fitting unconditional stability

dal19,carroll11
[12, 13] is the **Carroll** model given by

$$
\begin{aligned}
\psi^D &= \alpha\bar{I}_1 + \beta\bar{I}_1^4 + \gamma\bar{I}_2^{\frac{1}{2}} = \alpha I_1 I_3^{-1/3} + \beta I_1^4 I_3^{-4/3} + \gamma I_2^{1/2} I_3^{-1/3} \\
\frac{\partial\psi^D}{\partial I_i} &= \left\{ \alpha I_3^{-1/3} + 4\beta I_1^3 I_3^{-4/3} \quad \frac{\gamma I_2^{-1/2} I_3^{-1/3}}{2} \quad -\frac{\alpha I_1 I_3^{-4/3}}{3} - \frac{4\beta I_1^4 I_3^{-7/3}}{3} - \frac{\gamma I_2^{1/2} I_3^{-4/3}}{3} \right\} \\
\frac{\partial^2\psi^D}{\partial I_i \partial I_j} &= \begin{bmatrix} 12\beta I_1^2 I_3^{-4/3} & 0 & -\frac{\alpha I_3^{-4/3}}{3} - \frac{16\beta I_1^3 I_3^{-7/3}}{3} \\ 0 & -\frac{\gamma I_2^{-3/2} I_3^{-1/3}}{4} & -\frac{\gamma I_2^{-1/2} I_3^{-4/3}}{6} \\ -\frac{\alpha I_3^{-4/3}}{3} - \frac{16\beta I_1^3 I_3^{-7/3}}{3} & \frac{\gamma I_2^{-1/2} I_3^{-4/3}}{6} & \frac{4\alpha I_1 I_3^{-7/3}}{9} + \frac{28\beta I_1^4 I_3^{-10/3}}{9} + \frac{4\gamma I_2^{1/2} I_3^{-7/3}}{9} \end{bmatrix}
\end{aligned}
\tag{1.75}
$$

As the proof for the stability of invariant based models is not straightforward, the use of potentials associated with *principal strains* λ_i (equivalently, the eigenvalues of the matrix C) ensures the stability, as the matrix C is symmetric (note that these correspond to the square of the singular values of F). The Ogden model marckmann06 [11] is the most used model of this type

$$
\psi^D = \sum_{n=1}^{N} \frac{\mu_n}{\alpha_n} (\lambda_1^{\alpha_n} + \lambda_2^{\alpha_n} + \lambda_3^{\alpha_n}) \tag{1.76}
$$

This open form ensures that the model is able to fit any curve, but it comes at the cost of a large number of parameters and the need to compute eigenvalues at each step.
So called *physic based* potentials are based on assumptions of how the elastomer behaves at a microscopic level. The main origin of this type of model is the chain model marckmann06 [11], which assumes that the elasticity of networked chains is due to the entropic changes, thus it may be determined by the number of possible states. This chain model gave origin to the widely used Arruda-Boyce model, which assumes that the chains are distributed in a form that 8 chains are attached to the center and to the vertices of a cube. More recently the assumption that the chain is restricted to a tube gave origin to the tube model, and finally the assumption that these chains are continually distributed in a sphere gave origin to the sphere models which are very good at capturing the behavior with few parameters.
m_hyper implements basic checks of material properties as shown in the examples below. This allows verification of how κ controls how close ν is to the fully incompressible value 0.5.

```
m_hyper('urn','PadC{2.5264,0,0,120,3,rho1n,tyMoon,unTM}');
m_hyper('urn','PadA{2.5264,-0.9177,0.4711,1200,3,f .35,g .5688,rho1n,tyYeoh,unTM}')
```

s*hyperup

1.4.7 Mixed U-P elements small deformation

For nearly incompressible materials one uses mixed elements that interpolate displacement and pressure using different shape functions (see zienkiewicz_1989 [14] section 12.5, R3.06.08 [15] or Abaqus theory manual *Section*

3.2.3 Hybrid incompressible solid element formulation). These are called mixed U-P elements. In SDT switch to this formulation is done for rule 20002 (hexa20 for u, hexa8 for p/J with 8 Gauss point, similarly tetra10/tetra4 with 4 Gauss point, penta15/penta6 xxx, see) For small deformation, the mixed formulation uses a generalized strain of the form

$$\begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \\ p \end{Bmatrix} = \begin{bmatrix} N,x & 0 & 0 & 0 \\ 0 & N,y & 0 & 0 \\ 0 & 0 & N,z & 0 \\ 0 & N,z & N,y & 0 \\ N,z & 0 & N,x & 0 \\ N,y & N,x & 0 & 0 \\ 0 & 0 & 0 & N_p \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \\ p \end{Bmatrix} \tag{1.77}$$

The constitutive law is then split in deviatoric and pressure/isotropic/isochore parts using the expression of the deviatoric part of the strain $\{\epsilon_{dev}\} = \left[D_0 - \frac{2}{3}\{m\}\{m\}^T\right]\{\epsilon\}$, where $\{m\}^T = \begin{bmatrix}1 & 1 & 1 & 0 & 0 & 0\end{bmatrix}$ and $D_0 = diag\left(\begin{bmatrix}2 & 2 & 2 & 1 & 1 & 1\end{bmatrix}\right)$, leading to

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ \sigma_{yz} \\ \sigma_{zx} \\ \sigma_{xy} \\ s_p \end{Bmatrix} = \begin{bmatrix} [D]\left[D_0 - \frac{2}{3}\{m\}\{m\}^T\right] & \{m\} \\ \{m\}^T & -1/K \end{bmatrix} \begin{Bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \\ p \end{Bmatrix} \tag{1.78}$$

the virtual work associated with s_p is assumed to be equal to zero, so that one has the weak set of equations $0 = \int_\Omega N_p(p/K - \{m\}^T\{\epsilon\}) = \left[K_{qp}^T\right]\{q\} + K_{pp}\{p\}$ (which is avoids numerical problems by considering a p interpolation that is coarser than the displacement interpolation). When doing transients, the equations of motion would be of the form

$$\begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix} \begin{Bmatrix} \ddot{q} \\ 0 \end{Bmatrix} + \begin{bmatrix} K_{qq} & K_{qp} \\ K_{qp}^T & K_{pp} \end{bmatrix} \begin{Bmatrix} q \\ p \end{Bmatrix} = \begin{Bmatrix} F_{ext} \\ 0 \end{Bmatrix} \tag{1.79}$$

where the augmentation by pressure DOF is not a dynamic equation but a set of constraints.
As a reminder, in the isotropic elasticity case, one has $K = E/(3(1-2\nu))$, $G = E/(2(1+\nu))$ and

$$D = K \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \frac{2G}{3} \begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3/2 \end{bmatrix} \tag{1.80}$$

eq*

For cross checking of large deformation, one considers the **zero deformation** case. One then has $I_1 = I_2 = 3, I_3 = 1$, thus $\partial\psi^I/\partial I_3$ is vanishing and $\partial^2\psi^I/\partial I_3^2 = \kappa/4$, which applying (1.69) is consistent with (1.80). For the Yeoh model one has the isochore part

$$\frac{\partial^2\psi^D}{\partial I_i^2} = C_{10}\begin{bmatrix} 0 & 0 & -1/3 \\ 0 & 0 & 0 \\ -1/3 & 0 & 4/3 \end{bmatrix}, \qquad \frac{\partial\psi^D}{\partial I_i} = C_{10}\left\{1 \quad 0 \quad -1\right\} \tag{1.81}$$

and the deviatoric contribution

$$\begin{aligned} D^D &= 4\begin{bmatrix} 1 & 1 & 1 & & & \\ 1 & 1 & 1 & & & \\ 1 & 1 & 1 & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}\left(\frac{\partial^2\psi^D}{\partial I_3^2} + 2\frac{\partial^2\psi^D}{\partial I_2 I_3}\right) + 4\begin{bmatrix} 0 & 1 & 1 & & & \\ 1 & 0 & 1 & & & \\ 1 & 1 & 0 & & & \\ & & & -1/2 & & \\ & & & & -1/2 & \\ & & & & & -1/2 \end{bmatrix}\left(\frac{\partial\psi}{\partial I_2} + \frac{\partial\psi}{\partial I_3}\right) \\ &= \frac{4C_{10}}{3}\left(\begin{bmatrix} 2 & 2 & 2 & & & \\ 2 & 2 & 2 & & & \\ 2 & 2 & 2 & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix} - 3\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1/2 \end{bmatrix}\right) \end{aligned} \tag{1.82}$$

zhuravlev_2017

leading to $2C_{10} = G$ (as also said in [16] table 2.2).

perupcheck

1.4.8 Cross check examples

Incompressible cube with edge length L is considered as a first example. The static problem is

$$\begin{aligned} div\,\boldsymbol{\sigma} &= \mathbf{0} \\ div\,\mathbf{u} = tr(\boldsymbol{\varepsilon}) &= 0 \\ \boldsymbol{\sigma} &= -p\mathbf{I} + 2G\boldsymbol{\varepsilon} \\ u_x(x=0,y,z) = u_y(x,y=0,z) = u_z(x,y,z=0) &= 0 \\ \sigma_{xx}(x=L,y,z) &= P_{ext} \end{aligned} \tag{1.83}$$

The incompressibility condition yields $\epsilon_{yy} = \epsilon_{zz} = -\frac{\epsilon_{xx}}{2}$. Assuming linear displacements, boundary conditions leads to the system

$$\begin{cases} -p + 2Ga = P_{ext} \\ -p - Ga = 0 \end{cases} \tag{1.84}$$

Hence, we deduce that

$$u_x = \frac{P_{ext}}{3G}\,x, \quad u_y = -\frac{P_{ext}}{6G}\,y, \quad u_z = -\frac{P_{ext}}{6G}\,z \quad p = -\frac{P_{ext}}{3} \tag{1.85}$$

Nearly compressible cube is a second example. The static problem is

$$\begin{aligned} div\,\boldsymbol{\sigma} &= \mathbf{0} \\ p + K div\,\mathbf{u} &= 0 \\ u_x(x=0,y,z) = u_y(x,y=0,z) = u_z(x,y,z=0) &= 0 \\ \sigma_{xx}(x=L,y,z) &= P_{ext} \end{aligned} \tag{1.86}$$

The constitutive law can be rewritten as

$$\boldsymbol{\sigma} = \lambda tr(\boldsymbol{\varepsilon})\mathbf{I} + 2G\boldsymbol{\varepsilon} = K tr(\boldsymbol{\varepsilon})\mathbf{I} + 2G\boldsymbol{\varepsilon}^{dev} \tag{1.87}$$

Assuming linear displacements, the deformation reads

$$\varepsilon_{xx} = a \quad \text{and} \quad \varepsilon_{yy} = \varepsilon_{zz} = -\nu a \tag{1.88}$$

so that

$$tr(\boldsymbol{\varepsilon}) = a(1-2\nu) \tag{1.89}$$

Boundary conditions lead to the following system

$$\begin{cases} a\left[\lambda(1-2\nu)+2G\right] = P_{ext} \\ a\left[\lambda(1-2\nu)-2G\nu\right] = 0 \end{cases} \tag{1.90}$$

Recalling that

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \quad G = \frac{E}{2(1+\nu)} \quad \text{and} \quad K = \frac{E}{3(1-2\nu)} \tag{1.91}$$

the system becomes

$$\begin{cases} Ea = P_{ext} \\ 0 = 0 \end{cases} \tag{1.92}$$

So, we deduce

$$u_x(x,y,z) = \frac{P_{ext}}{E}x, \quad u_y(x,y,z) = -\nu\frac{P_{ext}}{E}y, \quad u_z(x,y,z) = -\nu\frac{P_{ext}}{E}z \tag{1.93}$$

$$p(x,y,z) = -\frac{P_{ext}}{3} \tag{1.94}$$

Note that $E = 3G$ for incompressible materials so that this solution tends to the one obtained in the incompressible case.

s*hyupld

1.4.9 Mixed U-P elements large deformation

In the large deformation case, one considers a development similar to the two field case described in R3.06.08 [15]. The generalized strain combines the displacement gradient and the internal state g related to volume change/pressure. The second Piola-Kirchhoff stress is given by

$$S_{ij} = 2\frac{\partial I_k}{\partial C_{ij}}\frac{\partial \psi^D}{\partial I_k} + \frac{\partial I_3}{\partial C_{ij}}g \tag{1.95}$$

eq*

and g related to volume change by considering the weak relation

$$\int_\Omega \hat{g}(g - \frac{\partial \psi^I}{\partial J}) = 0 \tag{1.96}$$

where the potential is given by (1.71) where $\frac{\partial \psi^I}{\partial I_3} = \kappa(1-I_3^{-1/2})$ or (1.72) where $\frac{\partial \psi^I}{\partial I_3} = \frac{\kappa}{2}\left(I_3^{-1/2} - I_3^{-1}\right)$.
As a result the generalized stress components are obtained using

$$\begin{Bmatrix} S_{ij} \\ s_g \end{Bmatrix} = \begin{bmatrix} 2\partial I_k/\partial C_{ij} & 0 & 2\partial I_3/\partial C_{ij} \\ 0 & 1 & -1 \end{bmatrix} \begin{Bmatrix} \partial\psi^D/\partial I_k \\ \partial\psi^I/\partial I_3 \\ g \end{Bmatrix} \tag{1.97}$$

For time integration, considering a bulk viscosity xxx.

yper3dvisc

1.4.10 Viscoelastic implementations

penas21,these_penas

Since [17, 18] demonstrated that the stress relaxation with saturation was the most appropriate, this is the currently implemented model on the derivatives of the potential dWdI=$\{\partial\psi^D/\partial I_1 \quad \partial\psi^D/\partial I_2 \quad \partial\psi^D/$
The law (1.26) is used on the deviatoric components of $\partial\psi^D/\partial I_i$.
For the compression part $\partial\psi^I/\partial I_3$, which is much stiffer for many elastomers, a viscous model $\partial\psi^I = \partial\psi_0^I + (\kappa_v/\kappa)\partial\dot{\psi}_0^I$ is used leading to the increment equation

$$\partial\psi^I(t_{n+1}) = \partial\psi_0^I(t_{n+1})(1+\frac{k_v}{\kappa dt}) - \partial\psi_0^I(t_n)\frac{k_v}{\kappa dt} \tag{1.98}$$

note that the have a viscosity dominate compression behavior at a target frequency f_v in Hz you should use $k_v = \kappa/(2\pi f_v)$.

s*nltime

1.5 Transient solution of non-linear equations

s*hbmt1

1.5.1 Principles

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

$$[M_R]\{\ddot{q}_R(t)\} + [C_R]\{\dot{q}_R(t)\} + [K_R]\{q_R(s)\} = [b_R]\{u(t)\} \tag{1.99}$$

eq*

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

$$\{q\}_N = [T]_{N\times NR}\{q_R(s)\}_{NR} \tag{1.100}$$

eq*

This representation is fairly standard. The data structure representation within SDT is described in section 1.8.1 . SDT/FEMLink supports superelement import from various FEM codes and more details are given in section 2.4.1 for NASTRAN, section ?? for Abaqus, and section ?? 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 vermot_2010 [19] section 4.1.4) assuming a modal basis associated with the underlying linear system (discussed in section 1.5.3).
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^0_{n+1} and $\dot{q}^0_{n+1}$ are expressed as

$$\left\{\begin{array}{l} q^0_{n+1} = q_n + h\dot{q}_n + h^2(\frac{1}{2}-\beta)\ddot{q}_n \\ \\ \dot{q}^0_{n+1} = \dot{q}_n + h(1-\gamma)\ddot{q}_n \end{array}\right. \quad (1.101)$$ eq*

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

$$\left\{\begin{array}{l} \Delta q_{n+1} = q_{n+1} - q^0_{n+1} \\ \\ \dot{q}_{n+1} = \dot{q}^0_{n+1} + \frac{\gamma}{h\beta}\Delta q_{n+1} \\ \\ \ddot{q}_{n+1} = \frac{1}{\beta h^2}\Delta q_{n+1} \end{array}\right. \quad (1.102)$$ eq*

Provided solution q^k_{n+1}, the residue is defined as

$$r^{k+1}_{n+1} = [M]\,\ddot{q}^k_{n+1} + [C]\,\dot{q}^k_{n+1} + [K]\,q^k_{n+1} - f_{cn+1} - f_{NLn+1}(q^k_{n+1}, \dot{q}^k_{n+1}, t_{n+1}) \quad (1.103)$$ eq*

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

$$J = \left[\backslash \frac{1}{\beta h^2} + \frac{\gamma 2\zeta_j\omega_j}{\beta h} + \omega_j^2 \backslash \right] \quad (1.104)$$ eq*

For one step formulation see vermot_2010 [19] 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

$$\left\{\begin{array}{c} q_{n+1} \\ \dot{q}_{n+1} \end{array}\right\} = [E(h)]\left\{\begin{array}{c} q_n \\ \dot{q}_n \end{array}\right\} + \{f_{ch}\} \quad (1.105)$$ eq*

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

$$\left\{\begin{array}{l} q_{n+1} = q_n + h\dot{q}_n + h^2\beta\ddot{q}_{n+1} + h^2\left(\frac{1}{2}-\beta\right)\ddot{q}_n \\ \dot{q}_{n+1} = \dot{q}_n + h\gamma\ddot{q}_{n+1} + h\left(1-\gamma\right)\ddot{q}_n \\ M\ddot{q}_{n+1} + C\dot{q}_{n+1} + Kq_{n+1} = f_{cn+1} \\ M\ddot{q}_n + C\dot{q}_n + Kq_n = f_{cn} \end{array}\right. \quad (1.106)$$ eq*

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

$$\begin{bmatrix} M+h^2\beta K & h^2\beta C \\ h\gamma K & M+h\gamma C \end{bmatrix} \begin{Bmatrix} q_{n+1} \\ \dot q_{n+1} \end{Bmatrix} = \begin{bmatrix} M-h^2\left(\frac{1}{2}-\beta\right)K & hM-h^2\left(\frac{1}{2}-\beta\right)C \\ -h\left(1-\gamma\right)K & M-h\left(1-\gamma\right)C \end{bmatrix} \begin{Bmatrix} q_n \\ \dot q_n \end{Bmatrix} + \cdots$$
$$\begin{Bmatrix} h^2\beta f_{cn+1} + h^2\left(\frac{1}{2}-\beta\right) f_{cn} \\ h\gamma f_{cn+1} + h\left(1-\gamma\right) f_{cn} \end{Bmatrix} \tag{1.107}$$

eq*

The evolution matrix is then

$$[E(h)] = \begin{bmatrix} M+h^2\beta K & h^2\beta C \\ h\gamma K & M+h\gamma C \end{bmatrix}^{-1} \begin{bmatrix} M-h^2\left(\frac{1}{2}-\beta\right)K & hM-h^2\left(\frac{1}{2}-\beta\right)C \\ -h\left(1-\gamma\right)K & M-h\left(1-\gamma\right)C \end{bmatrix} \tag{1.108}$$

eq*

and the interpolated external force is then

$$\{f_{ch}\} = \begin{bmatrix} M+h^2\beta K & h^2\beta C \\ h\gamma K & M+h\gamma C \end{bmatrix}^{-1} \begin{Bmatrix} h^2\beta f_{cn+1} + h^2\left(\frac{1}{2}-\beta\right) f_{cn} \\ h\gamma f_{cn+1} + h\left(1-\gamma\right) f_{cn} \end{Bmatrix} \tag{1.109}$$

eq*

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

$$\ddot q_{n+1} = \frac{1}{h\gamma}\left(\dot q_{n+1} - \dot q_n\right) - \frac{1-\gamma}{\gamma}\ddot q_n \tag{1.110}$$

eq*

s*nlDofSet

1.5.2 Enforced displacement, resultants

When considering enforced displacement, the residual at a given time is

$$\begin{Bmatrix} R_I \\ R_C \end{Bmatrix} = \begin{bmatrix} M_{II} & M_{IC} \\ M_{CI} & M_{CC} \end{bmatrix} \begin{Bmatrix} \ddot q_I(t) \\ \ddot q_C(t) \end{Bmatrix} + \left\{ F_{NL}\left(\begin{Bmatrix} q_I \\ q_C \end{Bmatrix}, \begin{Bmatrix} \dot q_I \\ \dot q_C \end{Bmatrix}\right)\right\} - [b_{Ext}]\{u(t)\} \tag{1.111}$$

eq*

Different strategies are implemented to compute the enforced time derivatives of q_I. Simple approaches uses the base definition of curves for each column of `bset.def` which corresponds to

$$\{\ddot q_I \quad \dot q_I \quad q_I\} = \{T_I\}\{\ddot u_I(t) \quad \dot u_I \quad u_I\} \tag{1.112}$$

eq*

Observation of the residual loads on the enforced part requires the R_I part of the residual to be computed, this prevents the use of an elimination strategy before residual computations and the elimination must thus be performed, when solving for q_c and its time derivatives. Resultant on a fixed body associated with the sum of an arbitrary set of non-linearities thus requires an enforced zero displacement (`DofSet` and not `FixDof` entry).
Non-linearities may also lead to resultant like observations (the transmitted load), but this is then considered to be a generalized form of stress so that the quantity of interest must be exported as a stress or an internal state during time integration.

s*hbmt2

1.5.3 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 1.6 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.

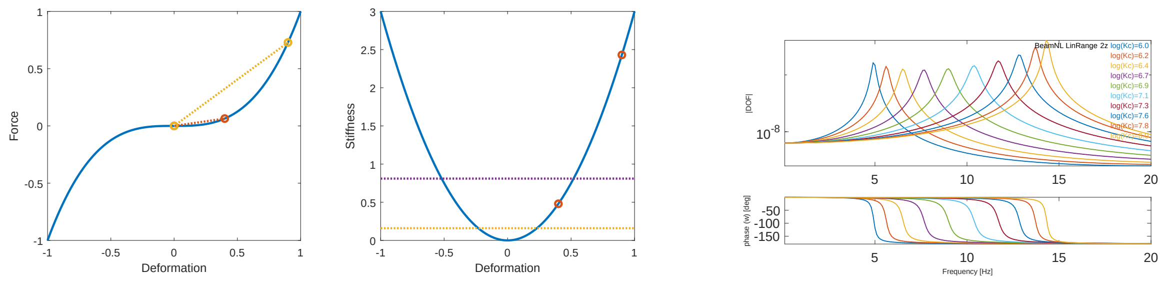


Figure 1.6: Left : Tangent and secant stiffness. Right : possible underlying linear systems for the BeamNL example.

:tanstiff

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

$$\{s_{nl}(t)\} = \{F_{nl}(e(t), \dot{e}(t))\} - [k_J]\{e_{nl}(t)\} - \{F_0\} \tag{1.113}$$

eq*

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.

s*nldata

1.6 Data structures for non-linearities in time

*nlformpro

1.6.1 NLdata non-linearity definition (model declaration)

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 1.3.7 , user defined anonymous functions see section 1.3.6 , tabular definitions section 1.3.2 .
- `.Fv` can be used for tabular definitions section 1.3.2 of laws on velocity.
- `.adof` optional definition of DOF specific to the non linearity that will be propagated to `model.FNLDOF`. For vectorized non-linearities, this gives the decimal part describing fields associated with `.snl`. To associate other DOF to `.unl`, you can set a `.udof` field.
- `.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. For the cell array format, rows correspond to each observed strain, and multiple columns can be given when multiple Gauss points differ. More details are given under `DOF representation of internal states`.
- `.keepLin` set to zero to avoid assembly of linear behavior in the base matrices
- `.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 (see `nl_spring NLJacobianUpdate`). Thus `MatType=1` stiffness corresponds to a displacement DOF, `2` mass an acceleration DOF, `3` viscous damping a velocity DOF.
- `.Jacobian` can be a function computing the tangent matrix or a number as described for `nl_spring nl_inout`.
- `.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.
- `.MexCb` is a cell array giving callback function and options (for an example see `d_fetime K_t`) . `.MexCb{1}=fun` should be a function handle within a function that also contains
  - `funtoOpt` the method checking init of the non-linearity, called in `nlutil('genToOpt')`

- funcheckFu post buildBC checks
- funJacobian implements Jacobian computation.

- .unl(:,:,2)=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 unl0='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.

modeltime

1.6.2 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 1.6.3 .
- .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`.

lformtime

1.6.3 `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 (1.3) 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 to `NL.unl` if buffer overwrite is acceptable) 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 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) dt0(3) val1 ...`
  - `val1, ...` is a `FuCode` dependent storage
  - for tables the standard, see `sdtweb('d_fetime','stdFuToOpt')`) is to use `iopt[6:3:.]` to point to start of table in `.opt` stored, give `M` as `[(double)jcur,Val/Type,s, table]`
- `.iopt` int32 data vector containing non-linearity specific integers. `(1)FInd` C 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. Later format may be dependent on non-linearity, a few variants are documented below to improve standardization
  - `FuTable` no internal state `iopt(6)=xStartC,(7)yStartC, (8)size(y,1), (9)size(y,2)` C style start in `.opt` of table with `unl` as first column and $F(u_{NL})$ values in second followed by `cur` (3 values). When combined with `FvTable` `iopt(10)=xStartC,(11)yStartC, (12)size(FvY,1), (13)size(FvY,2)`.
  - constitutive laws with internal states will use `iopt(6)StoreType` update strategy in residual call, `iopt(7:9)` formulation dependent choices. `iopt(10)=tstartC (11)size(y,1), (12)size(y,2)` may be used for tabular definitions (hyperelasticity for example) where table starts by 3 values `cur,val,s` followed by the x vector and y matrix.
- `.MexCb` is a cell array giving callback function and options, when calling MATLAB from the mex, . For optimized operation without fieldname checks, the option structure is assumed to have ordered fields :
  - `.unlg` at a single Gauss point,
  - `.opt` copy of `NL.opt`,
  - `.jg` int32 vector allowing passing of current Gauss point index,
  - `.vnlg` optional copy of velocities at gauss point or empty.
  - The `.mID` identifier can be used for C callbacks. See section 1.3.5 .
- 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]si`

- **obsolete**
- `.extDOF` obsolete see iopt, double vector containing Matlab indices `[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)`.

s*hbmsolv

1.7 Harmonic balance solutions and solver

s*hdof

1.7.1 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

$$\{f(t)\} = \sum_{k\in\mathcal{H}} f_k h_k(t) = \{f_k\}^T_{n_k} [H_{kt}]_{n_k\times n_t} \tag{1.114}$$

eq*

In the specific case of harmonic balance jaumouille11 [20], the time dependence of a given DOF is written as

$$q(t) = z_{cq0} + \sum_{n\in\mathcal{H}} \left(z_{sqn}\ sin\left(n\omega t\right) + z_{cqn}\ cos\left(n\omega t\right)\right) = \sum_{k\in\mathcal{H}} Z_{qk} h_k(t) \tag{1.115}$$

eq*

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

$$\{Z_{qk}\} = \left\{\begin{matrix} z_{cq0} \\ z_{sq1} \\ z_{cq1} \\ \vdots \end{matrix}\right\}_{n_q.n_h\times 1} \tag{1.116}$$

eq*

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 (1.128). 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 (1.115))
- `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

$$q(t_j) = Z_{ql} H_{lj} \tag{1.117}$$

eq*

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

$$H_{lj} = \begin{bmatrix} \mathbb{1}(t_1) & \dots & \mathbb{1}(t_N) \\ \sin(\omega t_1) & \dots & \sin(\omega t_N) \\ \cos(\omega t_1) & \dots & \cos(\omega t_N) \\ \vdots & \dots & \vdots \\ \sin(k\omega t_1) & \dots & \sin(k\omega t_N) \\ \cos(k\omega t_1) & \dots & \cos(k\omega t_N) \\ \vdots & \dots & \vdots \end{bmatrix} \tag{1.118}$$

eq*

In further equations and in the code, the notation `Hkt`(H_{kt}) is used even though in the real form (with `opt.Opt.complex=0`) there are two l lines for each harmonic k (except for harmonic 0). Note also that when using harmonic fractions (for example in engines it is usual to use $N/2$ harmonics to represent a period over two revolutions), you should declare `opt.Opt.nu=2` and the `.harm` field will contain $k = 1/2, 1, \ldots$.
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

$$Z_{ql} = \{q(t_j)\}_{Nq\times Nt}\,[H_{tk}]_{Nt\times Nt} = \{q(t_j)\}\,\frac{2}{N}\begin{bmatrix} \frac{1}{2} & \sin(\omega t_1) & \cos(\omega t_1) & \dots \\ \vdots & \vdots & \vdots & \dots \\ \frac{1}{2} & \sin(\omega t_N) & \cos(\omega t_N) & \dots \end{bmatrix} \tag{1.119}$$

eq*

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.

s*hbmbaset

1.7.2 Harmonic balance equation (real DOF)

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

$$\min_Z \|\{R(Z,\omega,u)\}\| = \min_Z \|[A(\omega)]\{Z\} - [b]\{u_{ext}\} - [b_{nl}]\{s_{nl}(Z,\omega)\}\| \tag{1.120}$$

eq*

The contents of matrices $[A(\omega)]$, $[b]$, and $[b_{nl}]$ depend on the choice of time functions in (1.117).
With the states described in (1.115) using (1.118), 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

$$R_0 = \sum_{i=1}^{N}(M\ddot{q} + C\dot{q} + Kq - F_{nl}(q,\dot{q},q_{nl},\omega,t_i) - F_{ext}(\omega,t_i)) \tag{1.121}$$

eq*

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

$$R_{sk} = \sum_{i=1}^{N}\sin(k\omega t_i)(M\ddot{q}(t_i) + C\dot{q}(t_i) + Kq(t_i) - F_{nl}(q,\dot{q},q_{nl},\omega,t_i) - F_{ext}(\omega,t_i)) \tag{1.122}$$

eq*

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

$$A_{\mathcal{H}} = \begin{bmatrix} K & & & & & & \\ & K-(\omega)^2 M & -\omega D & & & & \\ & \omega D & K-(\omega)^2 M & & & & \\ & & & \ddots & & & \\ & & & & K-(k\omega)^2 M & -k\omega D & \\ & & & & k\omega D & K-(k\omega)^2 M & \\ & & & & & & \ddots \end{bmatrix} \quad (1.123)$$ eq*

and a series of non-linear contributions

$$J_{\mathcal{H}} = [b_{\mathcal{H}}] \frac{\partial \{s_{\mathcal{H}}(Z,\omega)\}}{\partial Z} = [b_{\mathcal{H}}] \frac{\partial \{s_{\mathcal{H}}(\epsilon_{\mathcal{H}},\omega)\}}{\partial \epsilon_{\mathcal{H}}} \frac{\partial \{\epsilon\}}{\partial Z} = [b_{\mathcal{H}}] \frac{\partial \{s_{\mathcal{H}}\}}{\partial \epsilon_{\mathcal{H}}} [c_{\mathcal{H}}] \quad (1.124)$$ eq*

The linear part of $A_{\mathcal{H}}$ in (1.123) is skew-symmetric. The negative sign is applied to the lines corresponding to the sine contributions. Using the notation (1.115), the negative sign is then located on the upper triangular part.

In the implementations derived from ~~nlvibkit14~~ [21], 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 (1.120) 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

$$\{s_{\mathcal{H}}\} = [K_{\epsilon\mathcal{H}}] \{\epsilon_{\mathcal{H}}\} \quad (1.125)$$ eq*

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

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

$$K_{\epsilon\mathcal{H}} = \begin{bmatrix} k & & & & & & \\ & k & -k\eta & & & & \\ & k\eta & k & & & & \\ & & & \ddots & & & \\ & & & & k & -k\eta & \\ & & & & k\eta & k & \\ & & & & & & \ddots \end{bmatrix} \quad (1.126)$$ eq*

since $-i\sigma_s + \sigma_c = k(1+i\eta)(i\epsilon_s + \epsilon_c) = ik(\epsilon_s + \eta\epsilon_c) + k(\epsilon_c - \eta\epsilon_s)$.

Another form considers an extended Z with frequency stored as an additional component (this is achieved using `opt.Opt.fvar=1`)

$$\min_{Z_e} \|[A_e(\omega)]\{Z_e\} - b_e\| \tag{1.127}$$ eq*

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.

s*hbmkeq

1.7.3 Complex formulation and equivalent stiffness

To clarify the notion of equivalent stiffness found at the harmonic balance solution, the real harmonic balance states (1.115) can be written in complex form expressing the time response as

$$q(t) = \Re\left(\sum_{k\in\mathcal{H}} z_{qk} e^{ik\omega t}\right) \tag{1.128}$$ eq*

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

$$\{\mathcal{Z}_{qk}\} = \left\{\begin{matrix} z_{q0} \\ z_{q1} \\ z_{q2} \\ \vdots \end{matrix}\right\}_{n_q.n_h\times 1} \tag{1.129}$$ eq*

The harmonic function is then complex,

$$H_{lj} = e^{il\omega t_j} \tag{1.130}$$ eq*

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

$$z_{qk} = z_{cqk} - i z_{sqk} \tag{1.131}$$ eq*

as $e^{ik\omega t} = cos(k\omega t) + isin(k\omega t)$, the equivalence between (1.128) and (1.115) is indeed

$$\Re(z_{qk})cos(k\omega t) - \Im(z_{qk})sin(k\omega t) = z_{cqk}cos(k\omega t) + z_{sqk}sin(k\omega t) \tag{1.132}$$ eq*

For each harmonic k the mean equilibrium over a period (1.122) leads to an equation of the form

$$\left(-(k\omega)^2 M + ik\omega C + K\right)\{z_{qk}\} + \left(F_{nl}(q,\dot{q},q_{nl},\omega,t_i) - F_{ext}(\omega,t_i)\right)[H_{tk}] = 0 \tag{1.133}$$ eq*

where

$$F_{nl}(Z,\omega) = -[b]\, s_{nl} \tag{1.134}$$ eq*

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

$$s_k = s_{ck} - i s_{sk} = \mathcal{F}_k^{-1}\left(s_{nl}(Z,\omega,t_i)\right) = \sum_{i=1}^{N} \left(\cos(k\omega t_i) + i\sin(k\omega t_i)\right)\left(s_{nl}(Z,\omega,t_i)\right) \tag{1.135}$$ eq*

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

$$s_k = \mathcal{F}_k^{-1}\left(K(1+i\eta)u(Z,\omega,t)\right) = K(1+i\eta)u_k \tag{1.136}$$ eq*

Assuming a scalar stress relation, the non-linear harmonic problem (1.133) is thus equivalent to a linear harmonic problem

$$\left(-\omega^2 M + i\omega C + K + [b]\, K_{eq,k}\, [c]\right)\{\mathcal{Z}_k\} - F_{ext}(\omega) = 0 \tag{1.137}$$ eq*

with the equivalent complex stiffness given for each Gauss point by

$$K_{eq} = \frac{\mathcal{F}_k^{-1}\left(s_{nl}(Z,\omega,t_i)\right)}{u_k} \tag{1.138}$$ eq*

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

s*hbmjac

1.7.4 Inter-harmonic coupling discussion

Harmonic balance Jacobian estimation based on local non-linear physics is rarely discussed in the literature. Possible for a given harmonic to estimate an equivalent local linear constitutive law. This notion comes close to quasi-Newton 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 functional way (coherent with the existence of a constitutive law).

$$[J_{nl}] = \frac{\partial f_{nlj}}{\partial z_{hqk}} \tag{1.139}$$ eq*

$$f_{nlj} \simeq k_1\epsilon + k_2\epsilon^2 + k_3\epsilon^3 + \cdots \tag{1.140}$$ eq*

$$\epsilon \simeq \epsilon_{cq0} + \epsilon_{cq1}\cos(\omega t) + \epsilon_{sq1}\sin(\omega t) + \epsilon_{cq2}\cos(2\omega t) + \epsilon_{sq2}\sin(2\omega t) + \epsilon_{cq3}\cos(3\omega t) + \epsilon_{sq3}\sin(3\omega t) + \cdots \tag{1.141}$$ eq*

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

$$\left\{ \begin{array}{l} \left(k_1 - 9\epsilon_{cq1}^3 k_3\right)\delta_{cq1}\cos(\omega t) + \frac{3}{4}\epsilon_{cq1}^2 k_3 \delta_{cq1}\cos(3\omega t) \\ \left(k_1 + 9\epsilon_{sq1}^3 k_3\right)\delta_{sq1}\sin(\omega t) - \frac{3}{4}\epsilon_{sq1}^2 k_3 \delta_{sq1}\sin(3\omega t) \end{array} \right. \quad (1.142)$$

eq*

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

$$\left\{ \begin{array}{l} \left(k_1 - \frac{9}{4}\epsilon_{cq3}^2 k_3\right)\delta_{cq3}\cos(3\omega t) + \frac{3}{4}\epsilon_{cq3}^2 k_3 \delta_{cq3}\cos(9\omega t) \\ \left(k_1 + \frac{9}{4}\epsilon_{sq3}^2 k_3\right)\delta_{sq3}\sin(3\omega t) - \frac{3}{4}\epsilon_{sq3}^2 k_3 \delta_{sq3}\sin(9\omega t) \end{array} \right. \quad (1.143)$$

eq*

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

$$\left\{ \begin{array}{l} \epsilon_{cq1} k_2 \delta_{cq1} + k_1 \delta_{cq1}\cos(\omega t) + \epsilon_{cq1} k_2 \delta_{cq1}\cos(2\omega t) \\ \epsilon_{sq1} k_2 \delta_{sq1} + k_1 \delta_{sq1}\sin(\omega t) - \epsilon_{sq1} k_2 \delta_{sq1}\cos(2\omega t) \end{array} \right. \quad (1.144)$$

eq*

First order terms induced by the linear and quadratic constraint term respectively induced by $\epsilon = \epsilon + \delta_{cq3}\cos(2\omega t)$ and $\epsilon = \epsilon + \delta_{sq2}\sin(2\omega t)$

$$\left\{ \begin{array}{l} \epsilon_{cq2} k_2 \delta_{cq2} + k_1 \delta_{cq2}\cos(2\omega t) + \epsilon_{cq2} k_2 \delta_{cq2}\cos(4\omega t) \\ \epsilon_{sq2} k_2 \delta_{sq2} + k_1 \delta_{sq2}\sin(2\omega t) - \epsilon_{sq2} k_2 \delta_{sq2}\cos(4\omega t) \end{array} \right. \quad (1.145)$$

eq*

s*hbmload

1.7.5 Harmonic 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

$$\{F(t)\} = [b]\{u(t)\} \quad (1.146)$$

eq*

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

$$\{u_j(t)\} = u_{c0j}(\omega) + \sum_{k\in\mathcal{H}} u_{skj}(\omega)sin(k\omega t) + u_{ckj}(\omega)cos(k\omega t) \quad (1.147)$$

eq*

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

$$\{u_j(\omega)\} = \left\{ \begin{array}{c} u_{c0j}(\omega) \\ u_{s1j}(\omega) \\ u_{c1j}(\omega) \\ \vdots \end{array} \right\} \quad (1.148)$$

eq*

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 frequency 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 frequency `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 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 loading. 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,

$$\{u(\omega)\} = \begin{Bmatrix} \vdots \\ \{u_j(\omega)\} \\ \vdots \end{Bmatrix} \quad (1.149)$$ eq*

so that the total external harmonic load vector is expressed as

$$\{Z_f\}_{Nhdof \times 1} = [b]\{u(\omega)\} \quad (1.150)$$ eq*

datastruct

1.8 Data structures for HBM solvers

s*model

1.8.1 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.

`Opt`

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 second row of `Opt` specifies a matrix stored in the field `K{i}` (where `i` is the column number). The value of `Opt(2,i)` indicates the matrix type of `K{i}`. `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.

s*NLdata

1.8.2 Non-linearity definition NLdata

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

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. 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^{th} 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]`.

s*NL

1.8.3 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.

s*HbmOpt

1.8.4 Solver options definition

```
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.

- `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 polynomial 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.
  - `nu` (`= 1`) Defines the harmonic factor $\omega = k/\nu$.
  - `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 polynomials (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 every time 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.

mOptSolve

1.8.5 Option structure during HBM solve

Inside the solver loop, developers 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 (1.123) (this matrix gets overwritten during iterations).

- `opt.Hkt` unit evolution of a given harmonic.(1.118)
- `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.
- `opt.harm` internal field of retained harmonics.

curveIntro

1.8.6 Harmonic result structure

`Zcurve` 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.

```
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)))
```

s*hbm_tuto

Tutorial

Contents

m_install

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

- 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.

s*hbmf

2.2 Frequency domain test cases

s*hbmf_1

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 hammami_2014a [22].
- 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 1.2.3 .
- d_hbm('TestSPlate') illustrates a plate on non-linear supports.
- t_hbm TestLapJoint Bj2 (provided by AGI as part of project CLIMA)

s*hbmf_2

2.2.2 Spring mass examples

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

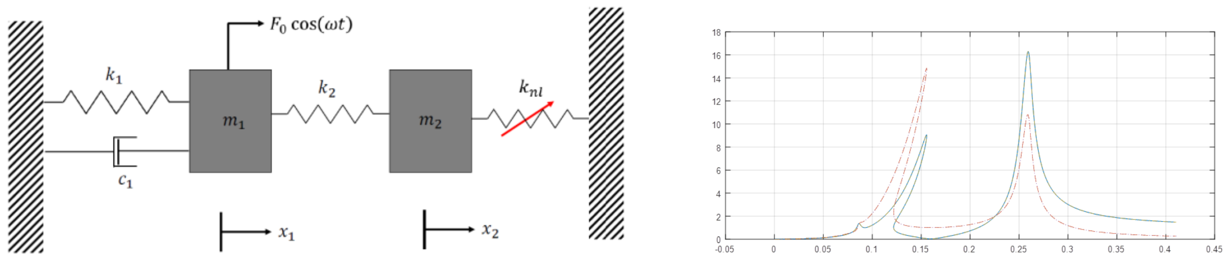


Figure 2.1: Two DOF duffing oscillator with cubic non-linearity

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

s*hbmf_3

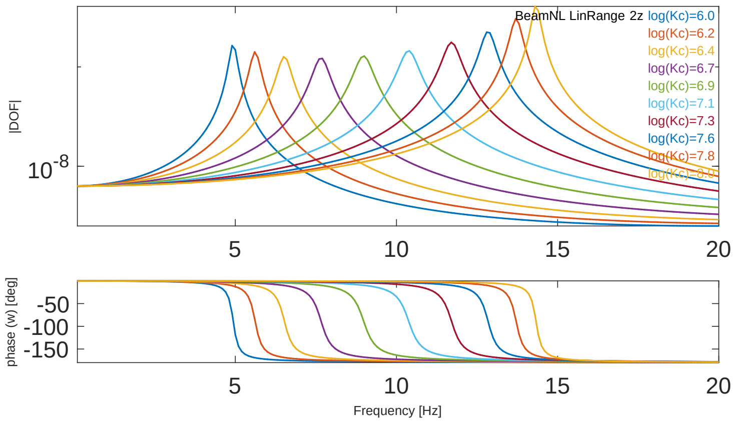
2.2.3 Beam problem with local non-linearity

```
sdtweb t_hbm beamnl
```

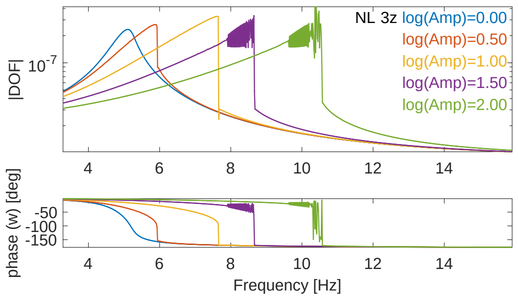
As a first example `clima16('BeamNLKrange')`, one seeks to demonstrate the sensitivity to a variable stiffness. The non-linear stress strain relation is defined by

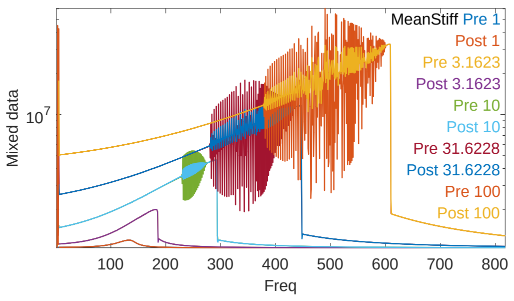
```
Fu=@(NL,fc,model,u,v,a,opt,Case,RO)kcur.*(NL.unl-loss*NL.vnl/opt.w);
```

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 $s = Re\left(k(1+i\eta)e^{i\omega t}\right) = Re\left((k + i\omega c e^{i\omega t})\right)$ 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 [22] for details).



As a second example `clima16('BeamNLARange')`, one seeks to illustrate the amplitude dependence obtained for a stiffening spring.





s*hbmf_4

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')`.

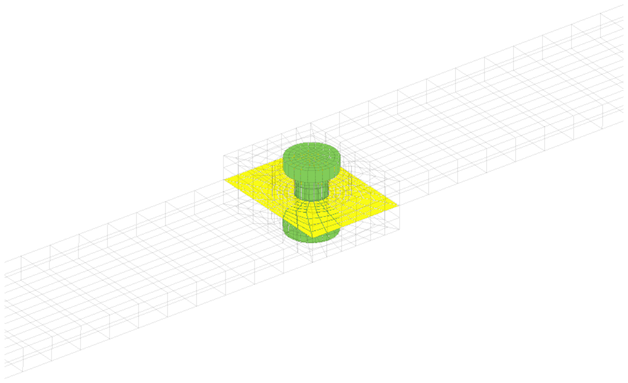


Figure 2.2: f:1j2 Lap joint with contact surface

s*hbmf_5

2.2.5 Hyperelastic bushing

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

```
[mo1,hopt]=d_hbm('TestBeamVNL');
```

s*hbmt

2.3 Time domain test cases

- sdtweb('_eval','d_fetime.m#BumpStop') simple mass on spring with bumpstop non-linearity.

s*hbmt_1

2.3.1 Single mass test of various non-linearities

Single mass test of various non-linearities 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
 li={'MeshCfg{"d_fetime(1DOF):MaxwellA{q0}"}';';'
    'SimuCfg{ModalNewmark{.1m,.3,fc,chandle1}}'};
 RT=sdtm.range(struct,li);mdl=RT.nmap('CurModel');
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=d_fetime('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));
```

s*hbmt_1s

2.3.2 Single mass stepped sine

The d_hbm TestSteppedSine example illustrates the stepped sine procedure implemented with the ModalNewmark solver.
See also d_tdoexxx.
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.

bmt_cbush

2.3.3 CBush with orientation

The example is detailed in

d_fetime('tutoCbushOrient')

s*hbmt_2

2.3.4 Beam with non-linear rotation spring

First do a sweep

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

s*hbmt_3

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

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

eq*

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

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

eq*

Since this integration does not guarantee $|F_x(t)| <= |\mu F_z(t)|$, 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

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

eq*

Iwan model with Dahl cells. Data is *sigma* slope at no load, α shape parameter, μF_z normal load.

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

eq*

TimeRange

2.3.6 Parametric experiments in time

`nl_solve` handles experiments associated with a series of time computations defined using the `fe_range` format. Typical applications are frequency and amplitude stepping experiments associated with HBM testing. For examples see
The principle of the experiment is that at each design point parameters are first changed before starting a time computation followed by postprocessing steps. The expression for changing the parameter can be given manually as in the first `d_fetime` `doTimeRange` example where

```
R1.param.Zener_c1.SetFcn='NL.opt(10:11)=[1 -1]*10/500*val(j1);';
```

changes the constants in `NL.opt(10:11)`.
Other predefined set functions are

- `fin_coef` do a stepped scaling coefficient on the time step. The Jacobian is updated but the loading profile is kept constant. This is typical of harmonic testing.
- `A,A0` can be used to
- `m_coef` adjust density for quasi-static testing.

xxx

s*nlse

2.4 Elastic representation as superelements

The base documentation on superelement import is part of SDT, see `SeImport` .

s*hbm_nas

2.4.1 NASTRAN cards used for sensors/non-linearities

The NASTRAN equivalent of superelement notions discussed in section **??** are

eq*nlse_cb

- q_I interface DOF of are defined in NASTRAN using `Bset` cards. These are stored in SDT as a `DofSet` entry to the model.
- b_{res} the independent vectors used to generate residual loads and lead to additional shapes using the residual vector procedures of NASTRAN
  - point loads simply declared using the `USET,U6` card
  - relative loads simply can 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` $(C(f)+iD(f))e^{i\theta-2\pi f\tau}$
- T_C columns are associated to scalar DOFs called `QSET`. 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.
- $y = [c]\{q\}$ observation. Does not exist in NASTRAN documentation, but implemented exporting SPOINT for each observation component and an MPC for each row of the observation matrix. This is achieved by modifying the model using `fe_sens('MeshSensAsMPC'` prior to export.
- `rigid` is known as `RBE2` in NASTRAN.

- rbe3 is known as RBE3 in NASTRAN.

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

s*hbm_abqs

2.4.2 ABAQUS cards used for sensors/non-linearities

The Abaqus equivalent of superelement notions discussed in section **??** are

- q_I interface DOF xxx
- b_{res} the independent vectors used to generate residual loads are written as independent load cases using xxx SeWriteBres
```
*LOAD CASE, NAME=LC000001
*CLOAD, OP=MOD
1001, 1, 1
*END LOAD CASE
```
- *FREQUENCY, RESIDUAL MODES : computation of attachment modes, coherent wit first order corrections.
- *EQUATION : equivalent the SDT MPC definition with a direct constraint matrix declaration.
- *KINEMATIC COUPLING : equivalent of SDT rigid connections where the spring is connected to a master node with 6 DOF which enforce motion of a number of slave DOFs.
- *DISTRIBUTING COUPLING equivalent of SDT 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).

s*hbm_anss

2.4.3 ANSYS cards used for sensors/non-linearities

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

2.4.4 Storing advanced SDT options in bulk format

For `upcom` parameters, export is done using design variables.
SDT-NLSIM provides an harmonic definition mechanism (see `hdof`). Storage in NASTRAN bulk format is as follows

```
$   1  $$   2  $$   3  $$   4  $$   5  $$   6  $$   7  $$   8  $$   9  $
$ All DOFs with sin(omega t) and cos(omega t)
DTI     HDOF    1       ALL     123456  CS1     ENDREC
$ Gradual building of full list of DOFs
DTI     HDOF    1       N1      THRU    N2      123     S1      N3
        THRU    N4      1       C1      N5      123456  S1      ENDREC
```

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  $$   2  $$   3  $$   4  $$   5  $$   6  $$   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  $$   4  $$   5  $$   6  $$   7  $$   8  $$   9  $
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       SID     101     FORM    Amp     UN1     UN2
        Harmi   ACi     ASi     ENDREC
```

- `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.

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

- 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 referring to table entries in the bulk.

s*d_squeal

2.5 Squeal related examples

s*sqsig

2.5.1 Post-processing test signals

xxx : place viewSpec in iicom sig xxx

s*sqcea

2.5.2 CEA

Simple brake example

s*sqtime

2.5.3 NL time transient on reduced bases

xxx

*hbm_range

2.6 Advanced usage

hbm_rangeb

2.6.1 DOE & general organization 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
    'case'    'Case 1'        [1x1 struct]
   % 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
```

hbm_other

2.7 External links

References to external documents. In SDT use `sdtweb('ref')` to open the page.

- `ParamEdit` standard SDT parameter extraction `base/cingui`
- `fe_case` load and boundary conditions `base/fe_case`
- `fe_caseg` assembly and GUI complement `base/fe_caseg`
- `ConnectionScrew`, `Assemble` assembly and GUI complement `base/fe_caseg`
- `fe_ceig` complex mode computations `base/fe_ceig`
- `fe_eig` mode computations `base/fe_eig`

- fe_gmsh interface to GMSH base/fe_gmsh
- fe_load load building base/fe_load
- MatType time integration base/fe_mknl#MatType.
- fe_range handling of experiments base/fe_range
- staticNewton non-linear static computation base/fe_time, part of fe_time .
- newmark linear Newmark solver, NLNewmark nonlinear Newmark solverbase/fe_time.
- m_elastic material property function , m_elastic
- beam1 beam element function beam
- celas spring element function celas
- def field at DOF structure def
- feplot display mesh base/feplot
- zcoef weighted matrix list zcoef
- superelements SeImport are documented at base/SeImport
  - s*hbm_anscb ANSYS superelements base/SeImport#hbm_anscb
  - s*hbm_abqcb Abaqus superelements base/SeImport#hbm_abqcb
  - upcom superelement keeping element matrices base/upcom
- integrules integration base/integrules
- iiplot display curves base/iiplot
- lsutil level set utilities base/lsutil
- AddSet FEM utilities base/feutil
- feutilb fem utilities utilities base/feutilb
- p_super superelement properties base/p_super
- p_spring spring properties base/p_spring
- MBBryan large rotation base/vhandle_matrix#MBBryan.

3

Contact modeling, theory and implementation

nt_theory

Contents

cont_model

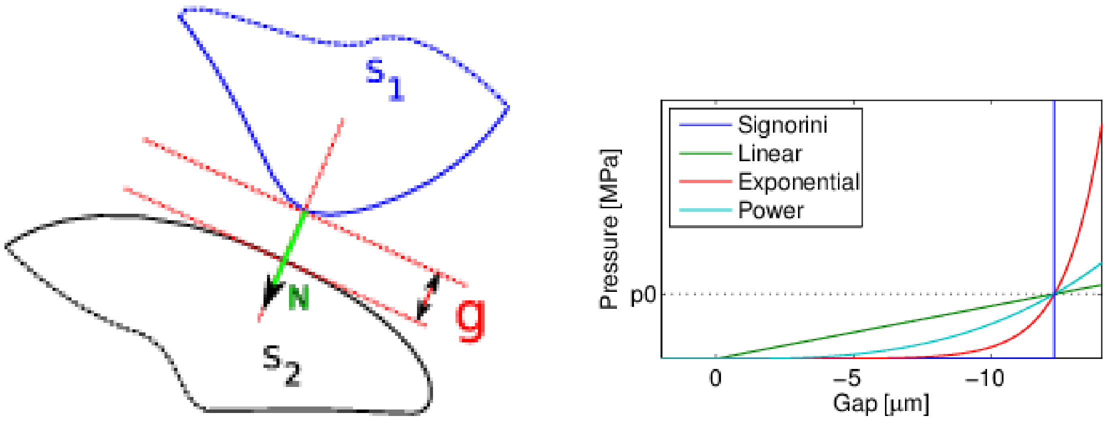
3.1 Modeling contact friction interactions

This section presents the two most classical ways of modeling contact friction interactions between structures. First physical models are described, then numerical implementation for finite element models is presented. Finally tangent contact friction derivation strategies are presented for modal computations.

CM_ContFri

3.1.1 Ideal Signorini-Coulomb model

The simplest contact representation – at least in formalism – is commonly known as the Signorini-Coulomb law to relate contact and friction. The definition of contact between to solids requires the notion of *gap* giving the distance between the solids considered, and taken as positive if non contact occurs. The notion of *contact normal* naturally follows as giving the way of measuring the gap. As presented in figure 3.1, the gap is measured as the shortest distance between two solids thus along the normal between both solids. It is function of the deformation u of each solid.



tFri_setup

Figure 3.1: Contact normal (N) and gap (g) definition for two solids, and the Signorini contact-pressure law.

In case of contact, a reaction force exists between both systems to avoid interpenetration. As contact between three dimensional solids is a surface, the notion of contact pressure is preferred here, and will be denoted p. It is of course directed by the contact normal. The Signorini conditions thus states for each point in contact

$$\left\{\begin{array}{r} g \geq 0 \\ p \geq 0 \\ (g).(p) = 0 \end{array}\right. \tag{3.1}$$

eq*

which sets exclusion between the gap values and contact pressure values. Two states are then possible, *opened* contact with a strictly positive gap and *closed* contact with a null gap and a strictly positive contact pressure. The law is plotted in figure 3.1. It is worth noting that Signorini contact is a mathematical idealization that is incompatible with the physics of surface irregularity at smaller scales.

The ideal Coulomb friction, as originally presented considers a so-called *static* friction model. The formulation distinguishes *sticking* and *sliding* states, depending on the existence of movement between the bodies in contact. If the solids in closed contact do not slide, a reaction force – the friction force – exists between both. This force is in the tangent plane defined as orthogonal to the contact normal N, and is determined by the equilibrium of the whole system. In the case of sliding, the friction force is opposed to the sliding velocity w, and only proportional to the normal load (or contact pressure) through the introduction of a *friction coefficient* μ.

Coulomb friction can be expressed as

$$\begin{cases} \|\{f_T\}\| \leq \mu\|\{f_N\}\| \\ \|\{f_T\}\| < \mu\|\{f_N\}\| \Leftrightarrow \{w\} = \{0\} \\ \|\{f_T\}\| = \mu\|\{f_N\}\| \Leftrightarrow \{f_T\} = \mu\|\{f_N\}\|\frac{\{w\}}{\|\{w\}\|} \end{cases} \tag{3.2}$$

where f_N is the contact force, f_T the friction force, μ the friction coefficient, and w the sliding velocity. The law is plotted in figure 3.2 which represents the friction force as function of the sliding velocity. For non-sliding states, the friction force is not defined by the law, as only an inequality is set.

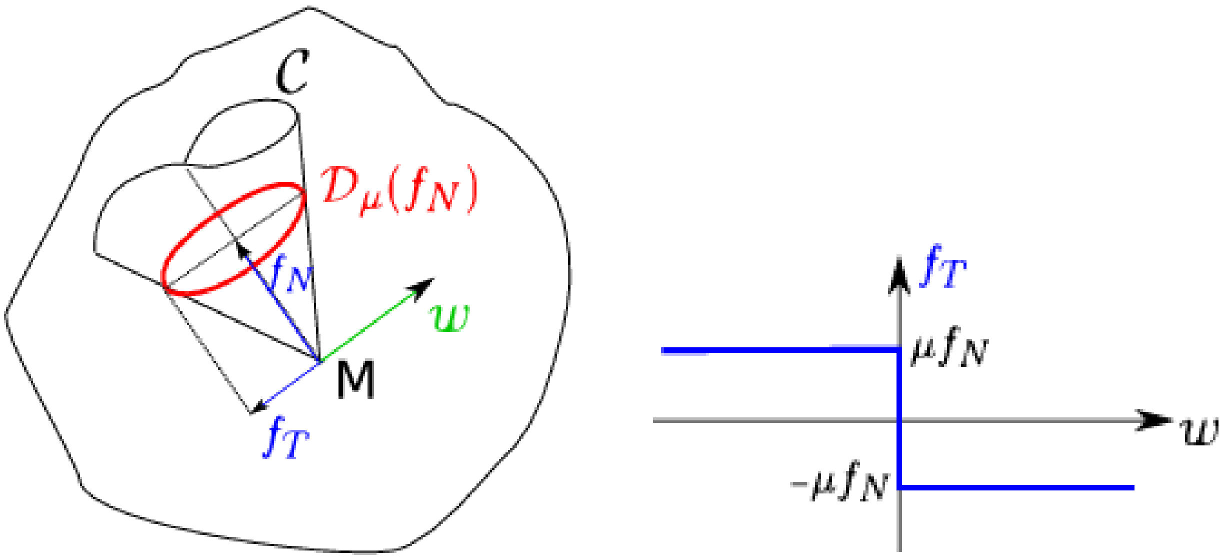


Figure 3.2: Ideal Coulomb friction law and representation of the Coulomb cone

The inequality concept is classically represented by the Coulomb cone shown in figure 3.2. For a given contact force at point M, the reaction force is said inside the coulomb cone $\mathcal{C}_\mu$, and more precisely in the cone section $\mathcal{D}_\mu$, determined by the tangent plane direction and the contact force amplitude.

The Signorini-Coulomb laws given by equations (3.1) and (3.2) raise numerical and analytical issues, since contact and friction forces are not defined for each point of their parameter. Indeed, for zero sliding velocity and zero gap, the forces are unknown and must be determined by other means.
To solve such problem a Lagrangian formulation has to be used, where the contact-friction forces are considered unknown and kept in the system to solve. The system size thus increases as function of the number of contact points, and features inequalities.

3.1.2 Functional representation of contact pressure

In addition to the numerical difficulties due to the Signorini-Coulomb formalism, the idealization represented may not always be satisfying. The reality of surfaces is indeed roughness describing the irregularities coming from the asperities of both surfaces [23]. This is conceptualized in figure 3.3.

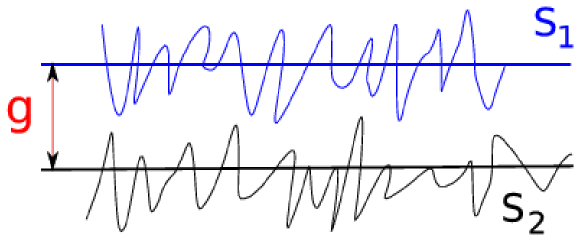


Figure 3.3: Illustration of roughness against nominal surfaces

Due to the very small scales of these asperities – around the micrometer – they are obviously not modeled in details. The roughness measures in particular commonly realize statistics on the whole surface, yielding a few parameters describing the surface state. Of course simulation of mechanical components consider nominal surfaces, with no asperities, as represented in figure 3.3.
In the world of asperities averaged by a nominal surface, interpenetration is possible at the macroscopic level, as function of the pressure applied on both surfaces. Indeed, the compression of asperities then increases allowing a greater contact pressure. From macroscopic tests, it is thus possible to obtain a law relating gap to contact pressure.
Possible functional representations of the pressure/gap relation are linear, power series or exponential laws, illustrated in figure 3.4.

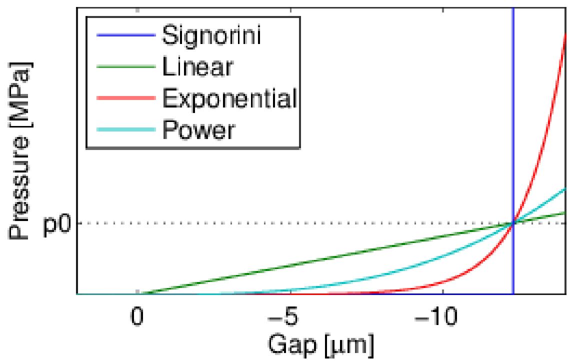


Figure 3.4: Classical contact laws, see section 3.3 for implementations

In practice, the contact pressure is applied to a deformable body. Assuming a certain height for this body, its stiffness k_e can be defined. The measurable displacement is related to pressure by

$$q = g(p) + p\frac{S}{k_e} \tag{3.3}$$

which is illustrated in figure 3.5.
Extreme regimes can easily be seen in equation (3.3). For a very thin body, the stiffness k_e is very high and the deformation will be mostly due to the deformation of asperities. In such case, a pressure/gap relation should be used.

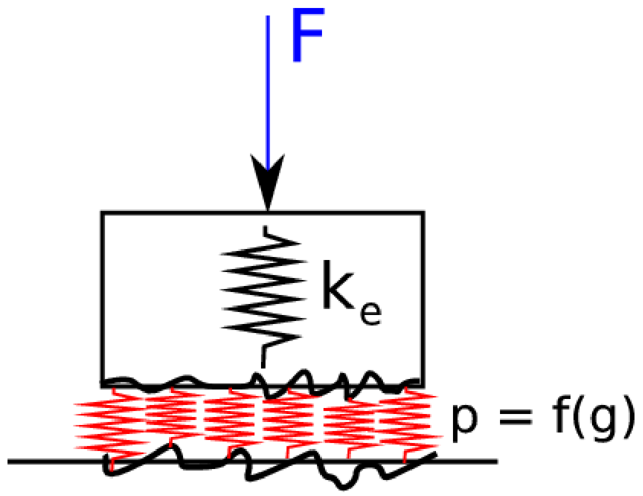


Figure 3.5: Representation of contact between two bodies, body stiffness and asperities

For many more configurations, the gap induced by realistic pressures will be small compared to deformations in the body. This regime is the reason for the widespread use of Signorini conditions. Since the real penetration is small, it cannot be measured accurately against body deformation and assuming it to be zero thus makes more sense. Furthermore, using it would cause major numerical difficulties since the accuracy on gap increments would need to be much higher than the accuracy desired for global motion.
Functional representations of contact pressures are also called penalization when the function is seen as a numerical tool rather than the result of an homogenization due to scale separation. In such case, gaps obtained in the simulation are (incorrectly) seen as numerical approximations. A relaxation of the Lagrange constraints is thus performed, authorizing interpenetration whose trade-off is the addition of a force linked to the violation of the constraint. The best interpretation is obtained energetically, where the relaxed constraint violation is generates a highly energetic term. Consequently, the mechanical solution will only show limited violations of the constraint.
It is noted here that the zero gap value that would be used in a Signorini model does not normally correspond to zero pressure, the contact of the mean smooth surfaces considered in this case corresponds to a state where some asperities are already into contact.
Using such laws, contact forces can be condensed on the displacement, making mechanical resolution much simpler. Contact law parameters must however be identified experimentally which may be difficult, whereas Signorini requires numerical adjustments. Contact mechanism is well assessed, and both Lagrangian and functional formulations have their adepts.

Practically the chosen contact model must be capable of properly transmitting pressure fields through the interface. To behave physically, the contact model should generate interpenetration at the expected scale and provide a coupling stiffer than the normal compression of the structure observed from the interface. If these aspects are respected, the structure will be able to see proper pressure fields at its interface.
It can be noted that non-linear functional representations, such as the exponential law (section 3.3) provide tangent stiffness fields varying with the gap, over the surface thus reflecting in the modal domain the effect of loading on the surface. Other laws, like the linear functional law or exact Signorini will provide the same tangent state whatever the contact pressure distribution if the contact surface remains identical. In such case, non-linear simulations are necessary to evaluate the effect of load variation over the surface.
One should also note that in finite element simulations, the appropriate model will depend on element size with, functional representations being more appropriate for small elements. More generally, computation with compatible meshes should be preferred. Contact simulation with non conforming interfaces can generate peculiarities that should require specific care, at least verification from the user.

CM_fricreg

3.1.3 Regularized friction models

Friction behavior is more complex, and the Coulomb law, as simple as it is, does not satisfy the needs of everybody.
On the first hand, regularization methods can be employed to relax the Coulomb law for zero sliding velocities. The idea is then to penalize low sliding velocities through the introduction of a slope at the origin, introducing a parameter k_t, so that

$$\begin{cases} \| \{f_T\} \| = k_t \| \{w\} \| & \text{if } \| \{w\} \| < \frac{\mu}{k_t} \| \{f_N\} \| \\ \| \{f_T\} \| = \mu \| \{f_N\} \| & \text{else} \end{cases} \tag{3.4}$$ eq*

as shown in figure 3.6.

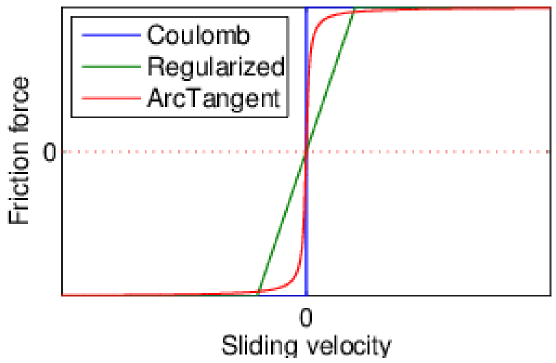


Figure 3.6: Sample Coulomb law regularizations fig:CF1_Coulomb_law_regularization

The basic regularization given by equation (3.4) is not perfect. First the threshold level will depend on the contact pressure
as its regularity is low, indeed, it is not differentiable at its transitions. An arctangent law can be used to alleviate the problem,

$$\| \{f_T\} \| = \frac{2\mu \| \{f_N\} \|}{\pi} \arctan\left(k_t \| \{w\} \|\right) \tag{3.5}$$ eq*

Two main drawbacks exist with such regularization. First, parameter k_t does not have a clear physical meaning, so that its value is difficult to determine. In particular, when dealing with distributed contact on large surfaces, the contact pressure will be shown to have major variations over the surface. The use of a constant k_t may thus be a problem.
Second, the regularized contact laws loose their static capability. In (3.4), a case with no sliding velocity implies no friction force. This is not compatible with stuck states when friction force is needed to maintain the sliding velocity to zero. Only stick-slip transition can then be approached for a globally sliding configuration.

s*cont_num

3.2 Numerical implementation

This section aims at presenting the algorithms implemented in the contact module for SDT.

s*cont_3d

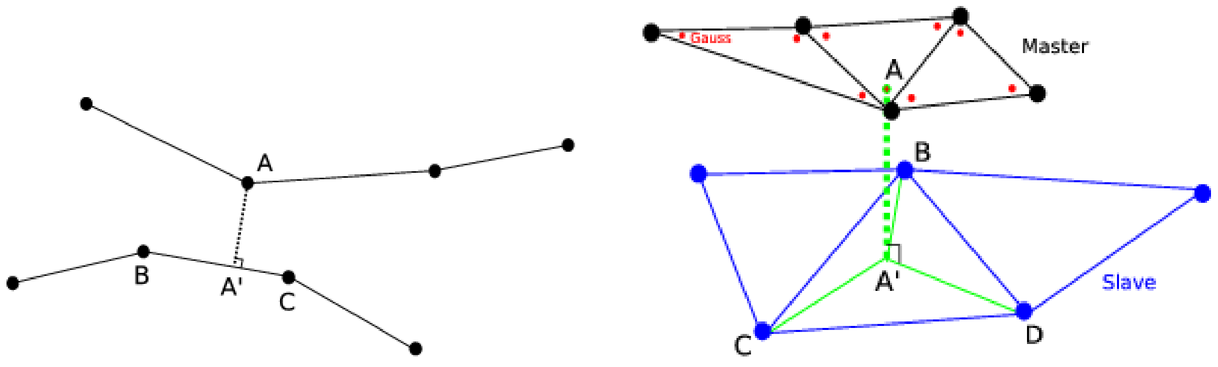
3.2.1 Contact friction between 3D surfaces

This section aims at presenting how the contact-friction models are be applied to a 3D finite element model using the contact module notations. The main idea to keep in mind is that the relevant contact information is the pressure field transmitted between surfaces.
The first feature to implement is the gap between two solids, as presented in figure 3.1. A slave/master strategy is commonly employed, so that the gap is evaluated by the projection of a contact point of the master surface onto the slave surface.
Practically, the gap between two surfaces is defined as the relative displacement along the contact normal N with a possible offset g_0, as

$$\{g\} = N.(u_{slave} - u_{master}) - \{g_0\} \tag{3.6}$$

eq*

For a 2D application, as illustrated in figure 3.2.1, contact point A is projected as A' on the slave surface, defining the gap as distance AA'. For a displacement u, the displacement of A' is expressed as a combination of the displacements of B and C, which are nodes of the slave surface model.



compute_2d

Figure 3.7: Illustration of gap computation for finite element models for 2D and 3D contact fig*cont_num_gapcompute

For a 3D application, as illustrated in figure 3.2.1, the computations become more complex in implementation. This case is illustrated in more details as it corresponds to the actual implementation in SDT contact module.
Once master and slave surfaces are identified, the so called *contact points* must be chosen. They will be the points where contact-friction forces will be actually computed for the mechanical equilibrium. In practice, they are points of the master surface, typically, the surface nodes are chosen, or in our case, the Gauss integration points of the underlying elements (red dots in figure 3.2.1). The

interest in choosing Gauss points for contact resides in the fact that an element-wise handling is permitted which allows usual integration rules to be employed and simplifies handling of surfaces for the definition of contact pressures.
The identification of the projected A' of contact point A is referred as the *pairing* operation, as each contact point of the master surface must be paired to the slave surface depending on its configuration. In practice, each contact point is first paired to an element of the slave surface, found as being the element in intersection with the outer normal of the contact point. The projected point A' is then expressed in position as a linear combination of the slave element points, B, C and D, which is commonly named *mapping.* Points B, C and D $\in \mathcal{N}_{\mathcal{S}e}$ the set of nodes of slave element e, they are computed using the shape functions of the slave element. They can be directly exploited, writing

$$u_{A'} = \sum_{x_k \in \mathcal{N}_{\mathcal{S}e}} \xi_k(A^{'}) u(x_k) \tag{3.7}$$ eq*

where $\xi_k(A^{'})$ are the coefficients relating the position of A' in e.
The gap defined by equation (3.6) can consequently be expressed for a contact point m_j as

$$g_j = N. \left(u(m_j) - \sum_{x_k \in \mathcal{N}_{\mathcal{S}e}} \xi_k(A^{'}) u(x_k) \right) - g_{j0} \tag{3.8}$$ eq*

In practice, the gap will be expressed at the contact points, but obtained from the mesh deformation u described by the DOF vector q. The practical equation to be exploited is then for a contact point j,

$$g_j = N. \left(\sum_{x_k \in \mathcal{N}_{\mathcal{M}e}} \left(\xi_k \left(A \right) q_k \right) - \sum_{x_k \in \mathcal{N}_{\mathcal{S}e}} \left(\xi_k(A^{'}) q_k \right) \right) - g_{j0} \tag{3.9}$$ eq*

where $\mathcal{N}_{\mathcal{M}e}$ (resp. $\mathcal{N}_{\mathcal{S}e}$) are the master element nodes containing (resp. the slave element nodes paired) with contact point j. $\xi_k(A)$ (resp. $\xi_k(A^{'})$) are the coefficients relating the position of contact point A in the master element (resp. of the projection $A^{'}$ of contact point A on the slave element). q_k is the displacement of node k.
Equation (3.9) is a linear combination of the system DOF, so that it can be summarized using an observation matrix $[C_{NOR}]$. The gap vector expressed at each contact point is then written

$$\{g\}_{N_c \times 1} = [C_{NOR}]_{N_c \times N} \{q(u)\}_{N \times 1} - \{g_0\}_{N_c \times 1} \tag{3.10}$$ eq*

noting N the number of system DOF, and N_c the number of contact points.
The generalized force resulting from the gap-pressure relationship, with the gap defined at each Gauss point by equation (3.10), is then defined as

$$\{\hat{q}\}^T \{f_N\} = \int_\Gamma \hat{u}(\hat{q}) N p dS \simeq \sum_e \sum_j \{\hat{u}(\hat{q})\}^T \{N\} p(x_j, q) \omega_j^{(e)} J^{(e)}(x_j) \tag{3.11}$$ eq*

where f_N is the generalized contact force, p the contact pressure, $\hat{q}$ a virtual displacement, q the displacement, $x_j^{(e)}$ are the integration points of current element e, $J^{(e)}(x_j)$ the Jacobian of the shape transformation (surface associated to each integration point) and $\omega_j^{(e)}$ the weighting associated with the integration rule of element e.
The use of observation matrix $[C_{NOR}]$ is also possible, so that in practice the generalized contact force is expressed as function of the local contact pressure as

$$\{f_N\}_{N\times 1} = [C_{NOR}]^T_{N\times N_c} \left\{\omega_j^{(e)} J^{(e)}(x_j)\ p(x_j, q)\right\}_{N_c\times 1} \tag{3.12}$$ eq*

The computation of the sliding velocity requires the computation of the differential velocities of the slave and master surfaces,

$$\{w\} = T.(\dot{u}_{slave} - \dot{u}_{master}) \tag{3.13}$$ eq*

where T represents the directions in the friction plane, defined as orthogonal to the contact normal N. T can consequently represent two directions, and the sliding velocity vector be of size $2N_c$. The same developments than for the computation of the gap can be performed, leading to the generation of a tangential displacement observation matrix $[C_{TAN}]$, from which the sliding velocity is

$$\{w\}_{2N_c\times 1} = [C_{TAN}]_{2N_c\times N}\ \{\dot{q}\}_{N\times 1} \tag{3.14}$$ eq*

and generalized friction forces can be recovered, from the local friction constraint f_{tj}

$$\{f_T\}_{N\times 1} = [C_{TAN}]^T_{N\times 2N_c} \left\{\omega_j^{(e)} J^{(e)}(x_j)\ f_{tj}(p(x_j, q))\right\}_{2N_c\times 1} \tag{3.15}$$ eq*

In practice the $[C_{TAN}]$ matrix is expressed sequentially on both local directions of the friction plane,

$$[C_{TAN}]_{N\times 2N_c} = \left[[C_{TAN}]_{1N\times N_c}\ [C_{TAN}]_{2N\times N_c}\right] \tag{3.16}$$ eq*

cont_euler

3.2.2 Eulerian sliding formulation

For applications where an externally driven motion exists, the relative displacement of slave and master surfaces becomes a combination of a rigid displacement and a vibration field. In the case where the enforced motion is in the main friction direction, it can be convenient to switch to an Euler description for the external motion. In such formulation, the material is considered to be moving inside a fixed mesh. Additional terms are thus added to the relative displacement observation. The significant benefit is that matching and observation matrices can be generated once.
In practice, a drive speed in the sliding motion direction is added to the relative velocity observation. For rotating devices, one considers a rotating local frame. The rotation effect must then be directly taken into account in the system mechanical equations, which will be referred as an *Eulerian* formulation. As proposed by Moirot moi1 [24], the mesh will remain fixed while the disc material rotates inside. Numerically, the continuous displacement u of the disc can be expressed in the rotating frame as

$$u = u(r, \theta(t), z, t) \tag{3.17}$$ eq*

The frame is considered to rotate at a constant velocity Ω, around the e_z direction. As the time span possible from the transient simulations provided in chapter **??** is in the order of the tens of milliseconds the effect of the disc slowing down is neglected, so that

$$\theta(t) = \theta + \Omega t \tag{3.18}$$

eq*

The velocity of the disc can then be derived from equation (**??**) considering the total derivative of u, which implies the variations of u as function of its time dependent parameters and the rotation of the base vectors,

$$v = \frac{du}{dt} = \frac{\partial u}{\partial t} + \frac{\partial u}{\partial \theta}\frac{\partial \theta}{\partial t} + \Omega e_z \wedge u \tag{3.19}$$

eq*

which can be simplified using (3.18),

$$v = \frac{\partial u}{\partial t} + \Omega \frac{\partial u}{\partial \theta} + \Omega e_z \wedge u \tag{3.20}$$

eq*

The sliding velocity is the difference between the disc velocity computed in the rotating frame in equation (3.20) and the velocity of the pad defined in the fixed frame, so that

$$w = v_{pad} - v_{disc} = \frac{\partial u_{pad}}{\partial t} - \frac{\partial u}{\partial t} - \Omega \frac{\partial u}{\partial \theta} - \Omega e_z \wedge u \tag{3.21}$$

eq*

The disc rotation provides a natural sliding direction. It was thus chosen in this work to ignore radial velocities. The projection of equation (3.21) on e_θ yields

$$w_\theta = \left(v_{pad} - v_{disc}\right) e_\theta - \left(\Omega \frac{\partial u}{\partial \theta}\right) e_\theta + \Omega u \cdot e_r \tag{3.22}$$

eq*

where the product $(e_z \wedge u) \cdot e_\theta$ from the projection of equation (3.21) on e_θ is simplified as the projection of the positions on the radial direction.
In (3.22), the sliding velocity can be decomposed in three contributions from left to right a vibration, convection, and rotation terms can be observed. The first one comes from the fixed mesh vibrations, which are actually computed on the system. The rotation term is constant and only depends on the mesh original topology. The convection term is more difficult to implement as it implies spatial derivations in cylindrical coordinates with a mesh in Cartesian coordinates. This term is thus ignored in [25]. lor1
Within an element, displacement u is related to degrees of freedom q through shape functions N

$$u = Nq \tag{3.23}$$

eq*

the convection term is derived as

$$\frac{\partial u}{\partial \theta} \cdot e_\theta = \frac{\partial N}{\partial x_j}(q \cdot e_\theta)\frac{\partial x_j}{\partial \theta} \tag{3.24}$$

eq*

where x_j represents the Cartesian directions.

In the contact implementation used for this work, shape function utilities of SDT are used to generate an observation matrix $[C_{N\theta}]$ relating convective contributions of the sliding velocity to the vector of DOF q as in equation (3.24).
In practice, the sliding velocity is thus computed using

$$\{w_\theta\}_{N_c\times 1} = [C_{TAN}]_{N_c\times N}\{\dot{q}\}_{N\times 1} + \Omega\,[C_{N\theta}]_{N_c\times 1}\{q\}_{N\times 1} + \Omega\,\{r\}_{N_c\times 1} \tag{3.25}$$ eq*

where the tangent displacement observation matrix $[C_{TAN}]$ has been projected on the tangential direction e_θ, and $\{r\}$ is the radius of each contact point to the rotating frame origin. The names use for generalized strain components are `wutt` for $\Omega\,[C_{N\theta}]_{N_c\times 1}\{q\}_{N\times 1}$ and `wrg` for $\Omega\,\{r\}_{N_c\times 1}$ is included in the `vnl(:,:,2)`.
The effect of the convection term is interesting to interpret, it accounts for the fact that the disc has a static *Eulerian* deformation due to friction. The disc slows down entering the pad area and accelerates when existing. The effect is obviously a linear function of the friction coefficient.

The disc rotation also has an effect on the expression of the acceleration for the formulation of the mechanical equilibrium. The acceleration of the disc is defined as

$$a = \frac{dv}{dt} \tag{3.26}$$ eq*

which is obtained by deriving equation (3.20),

$$a = \frac{\partial^2 u}{\partial t^2} + 2\Omega\left(\frac{\partial^2 u}{\partial t\partial\theta} + e_z \wedge \frac{\partial u}{\partial t}\right) + \Omega^2\left(\frac{\partial^2 u}{\partial \theta^2} + 2e_z \wedge \frac{\partial u}{\partial \theta}\right) - \Omega^2\,(u\cdot r) \tag{3.27}$$ eq*

where r is the projection vector on the tangent plane $\{e_r; e_\theta; 0\}$.
Equation (3.27) presents several additional terms to the displacement acceleration $\frac{\partial^2 u}{\partial t^2}$. From left to right, a gyroscopic damping associated to the first order time derivatives of u (2Ω factor), a gyroscopic stiffness associated to the θ only derivatives of u (Ω^2 factor) and a volume centrifugal load are identified.
The mechanical equilibrium equation should then be altered to stand for all rotational effects in the disc. In practice however, the disc rotation velocities targeted are rather small – a few rad/s – so that all gyroscopic and centrifugal terms are neglected, for squeal applications, thus leading to

$$a \simeq \frac{\partial^2 u}{\partial t^2} \tag{3.28}$$ eq*

Adjusting rotation related terms in the equilibrium equation is a perspective for future work.

3.2.3 Frequency domain linearization

When characterizing a non-linear system in the frequency domain, the most common approach is to consider the system linearized around a deformation state and compute the modes of the linearized system. The linearized model is then called *tangent* to the said deformation state. This strategy is of course limited, as linearizing in the vicinity of deformation states can only be relevant for very small deformations from the targeted working point.
The expression of the system linearization strongly depends on the chosen contact formulation. For Lagrangian formulations, the definition of the tangent state only depends on the surface effectively in contact, as a contact point is either *open* (no contact) or *closed* (in contact). The open case generates no contact stiffness, while the closed case generates an exact continuity constraint for the normal displacement at the interface.
Constitutive contact formulations are defined as function of the gap, such that a tangent state can be derived for each contact point by deriving the gap/pressure relation as illustrated in figure 3.8. It should be noted that the contact stiffness differs at each contact point and depends on the local gap.

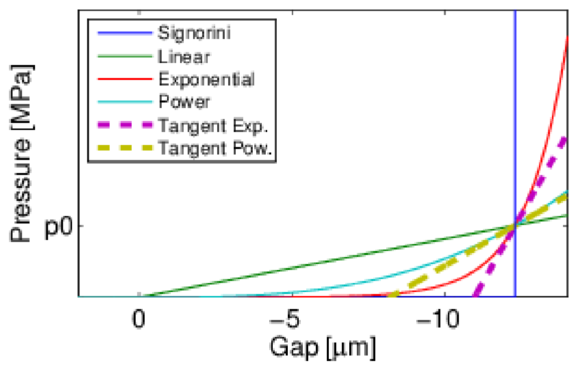


Figure 3.8: Illustration of tangent contact states at a given pressure state

At a given contact point i the contact stiffness k_{ci} is defined by the derivative of the pressure gap relationship,

$$k_{ci}(q) = \omega_i J(x_i) \frac{\partial p_i(q)}{\partial q} \tag{3.29}$$

where p_i is the pressure at contact point i, $J(x_i)$ the corresponding surface, and ω_i the Gauss weighting coefficient. Using the gap observation equation (3.10) and the force restitution of equation (3.12), the tangent contact stiffness matrix $K_{nlc}(q)$ is recovered,

$$K_{nlc}(q) = [C_{NOR}]^T \left[{}^{\backslash}\omega_i J(x_i) k_{ci}(q)_{\backslash} \right] [C_{NOR}] \tag{3.30}$$

K_{nlc} is therefore symmetric positive definite and can be used for real and complex modes.
The effect on the system itself is a perceived stiffness variation occurring at each point of the contacting interfaces. The distribution of contact stiffness can then show large variations over the surface.

The definition of a tangent friction state is more difficult, as it couples normal and tangential directions, it is typically non-symmetric – thus being a common source of instability.
The nature of friction as written in equation (3.2) shows that a variation of contact force has a direct effect on friction force, while a variation of the friction force may occur without effect on the contact force. This non symmetric notion is illustrated in figure 3.9 where stiffness coupling matrix topologies are illustrated for a basic case.

In the case of a sliding state, the friction force is explicitly defined and only depends on the contact force at the same point. The tangent friction is then the tangent contact (3.29) scaled by the friction coefficient μ, linking the normal and tangent displacements for complex modes. For real mode computation, since the tangent displacement is free for a fixed contact state, no stiffness is added. The tangent friction coupling stiffness K_{nlf}^{slide} for sliding states is defined as

$$K_{nlf}^{slide} = \begin{cases} 0 & \text{for real mode computation} \\ [C_{TAN}]^T \left[{}^{\backslash} \mu k_{ci}(q) {}_{\backslash}\right] [C_{NOR}] & \text{for complex mode computation} \end{cases} \tag{3.31}$$

eq*

The formulation of tangent friction matrices of equation (3.31) is very classical, and is commonly employed under the *small sliding perturbations* formulation. This was used by Moirot moi1 [24], Lorang lor1 [25] for brake squeal applications, or by Vola *et al.* vol01 [26] to study rubber/glass instabilities in sliding steady states.

Sticking friction state is very different in application. The basic Lagrangian approach will consider an exact continuity constraint of the tangential displacement at the interface, thus making it a bilateral constraint. Taking such pattern into account is critical for the complex modes as a physical resistance to the tangential displacement occurs. For sliding to happen, the friction force must increase up to the sliding threshold, this friction force variation must then be reproduced.
A functional approach can be used, which relaxes the Lagrange constraint. The friction force is however unknown inside the Coulomb cone, and further studies are necessary to characterize the stiffness coupling levels and the system sensitivity. Experiments are suggested by Bureau *et al.* bau2 [27] for non metallic materials. Ultrasonic measurements as done by Biwa *et al.* can also be a possibility biw01 [28]. In particular Biwa proposed a link between the sticking state stiffness and the interface material properties.
The strategy chosen here is to consider the Coulomb cone limit, characterized by the contact force level, illustrated in figure 3.2. A proportional relationship linking the tangent friction stiffness level and contact levels is set, using parameter κ. The stiffness behavior will thus be close to microslip patterns, occurring for low vibration levels. Obviously κ, which has to be somehow identified, will

verify

$$0 \leq \kappa \leq \mu_s \tag{3.32}$$

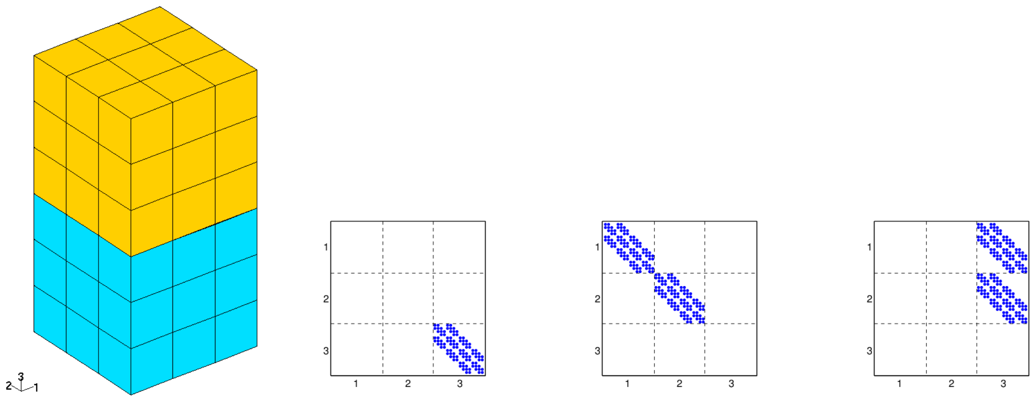
eq*

The tangent friction coupling stiffness K_{nlf}^{stick} for sticking states is then defined for both real and complex mode computation as

$$K_{nlf}^{stick}(q) = [C_{TAN}]^T \left[{}^{\backslash}\kappa \omega_i J(x_i) k_{ci}(q)_{\backslash} \right] [C_{TAN}] \tag{3.33}$$

eq*

To illustrate the effect of the different tangent contact-friction matrices presented here, an example based on a very simple problem is presented. The basic model given in figure 3.9 features two cubes in vertical contact. For illustration purposes, the degrees of freedom are segregated by their direction, 1 and 2 define the horizontal directions while 3 is the vertical normal direction, in which contact is defined.



ontactTOPO

Figure 3.9: Tangent contact-friction matrix topologies on a simple case. Direction 1,2 are horizontal (friction plane). Direction 3 is vertical (contact direction). From left to right, cube model, tangent contact coupling, sticking friction coupling and sliding friction coupling matrices

Figure 3.9 presents the topologies of the tangent contact-friction matrices expressed on the master surface DOF. The tangent contact matrix (3.30) couples the normal directions, here the third one, as shown in figure 3.9. The matrix is symmetric positive definite and used for real or complex mode computation.

The tangent friction matrix (3.33) for sticking states is plotted in figure 3.9. It couples directions 1 and 2 independently, is symmetric positive definite and is used in real and complex modes computation.

For sliding states, the tangent friction matrix (3.31) is shown in figure 3.9. The un-symmetric coupling is clearly observed as direction 3 is coupled to both directions 1 and 2. A deformation along 3 has an effect on directions 1 and 2 but not the other way around.

s*ScLd

3.2.4 Surface Contact with large displacement

Partial support of large displacement surface contact problems is being implemented for point (node or Gauss) to surface contact (look for the ScLd tag) . The generalized strain stored in unl is
xp,yp,zp, g,r,s,ielt,wjdet,n3x,n3y,n3z
corresponding to 3 strains and 8 internal states. The 3 strains are the current position of the contact point obtained as $x_p = x_{p0} + u_{xp}$. The gap is the first internal state and is counted as positive if the point is outside the surface (there actually is a gap rather than penetration). r, s give the position within current the triangular element of C index ielt. wjdet is the weight associated with the point needed to apply a load and n_{3i} are the components of the normal vector specifying the direction of the normal load.
The unl0=unl(:,:,2) stores xp0,wjdet,n3x,n3y,n3z which do not change. g of the previous time step which can be used for laws that use time derivatives of g. r, s, i_{elt} of the previous search which is used to speed search of the next point.
A residual call performs the following steps

- standard (pre Fu call) observation of the point position as $x_p = x_{p0} + u_{xp}$. In the case of Eulerian rotation. $v_p = [c]\{\dot{q}\} + \Omega\,[c_{N\theta}]\{q\} + \Omega r$
- non-standard (within the Fu call) observation of current element node positions .xyz_s and .cDofPos_s.
- search of positions within the triangular surface r, s and gap. The surface element is assumed to be oriented with an outgoing normal as standard in SDT, thus defining the gap sign.
- If point as moved out of current surface, use of .Adjacent_s table to update the matching surface.
- update of .b matrix indices and coefficients to correspond to a relative unit force between the node and the surface.
- compute 3 components .snl giving normal and potentially friction force.

s*LcLd

3.2.5 Line contact for large displacement

This unsupported development is found in oscar_panto BBeam.

cont_laws

3.3 Implemented contact and friction laws (SDT contact)

_laws_gen

General parameters

This functionality is distributed as part of the SDT contact. For general theory see section 3.1.2 .
The current `chandle` implementation (SDT ≥ 7.4) uses mex files and uses the convention of positive pressures for positive overclosure (opposite of gap $-g$). Observations of overclosures are thus of the usual form $-g = u_{nl} = [c]\{u\} + \{u_{nl0}\}$. For sliding, the convention of using tangential observation and commands with the same sign convention give contact forces with a minus sign.
For `chandle` runtime formatting, the implementation considers Gauss points with 3 gap strain components $g = u_N, u_{T1}, u_{T2}$ and possibly xxx (`unllab={'gn';'gt1';'gt2';'utt'}`).
For 3 strain velocities v_N, w_1, w_2 the formulation is given in (3.25)

$$\begin{Bmatrix} p_N \\ p_{t1} \\ p_{t2} \end{Bmatrix} = f\left(\begin{Bmatrix} u_N \\ u_{T1} \\ u_{T2} \end{Bmatrix}, \begin{Bmatrix} v_N \\ w_1 \\ w_2 \end{Bmatrix}\right) \tag{3.34}$$

eq*

Laws are selected using a URN combining normal (see `sdtm.enum('nlCtc')`) and tangent laws (see `sdtm.enum('nlSli')`) .

- `n3e13` tabular normal `n1` and arctan Coulomb `e13`. For non-tabular laws, constants used in the laws detailed below are ordered from `opt(10)` as `Kc lambda Mu CtLin kappa` (depending on the law only some parameters may be used). For tabular laws, the parameters can be used to build the table and are stored as fields of `NL.Fu`.
- need document of Eulerian rotation used for squeal and rail/wheel contact.

The `nlutil('@Ctc')` subfunction implements conversion of standard combinations of parameters to tabular laws

```
Ctc=nlutil('@Ctc');
Ctc('urn','n1s13{Kc1e12,Lambda2}')
```

Older versions use `.m` file implementation, gaps and not overclosures and the offsets, $g = [c]\{u\} + \{cq_{off}\}$ is stored in field `.cqoff`. Laws are selected using the `ContactLaw` value listed below.

_laws_lin

Bilinear contact law, `1,n1` : `linear`

The simplest contact representation is to apply a linear reaction force to contact points with negative gaps. The resulting law is bilinear, expressing contact pressure at a contact integration point i as

$$p(x_i) = \begin{cases} 0 & \text{if } g_i \geq 0 \\ k_c(-g_i) & \text{if } g_i < 0 \end{cases} \tag{3.35}$$

eq*

This law has a single parameter k_c (in Pa/m, or N/m^3).

The contact stiffness for tangent contact states can thus have two values depending on the gap at contact integration point i

$$k_{ci}(-g_i) = \begin{cases} 0 & \text{if } g_i \geq 0 \\ k_c & \text{if } g_i < 0 \end{cases} \tag{3.36}$$

eq*

k_c, of input name `Kc`, can be set to *`Val`* in `Fu` as a string of format `'Kc Val'` or as a scalar of value *`Val`*.

_laws_exp

Exponential contact law, `2` : `exponential`

The exponential contact law applies a non linear reaction force for all gaps. It is expressed at a contact integration point i as

$$p(x_i) = p_z e^{\lambda(-g_{ni})} \tag{3.37}$$

eq*

The pressure applied is strictly positive for all gaps, but its fast decay for low gap values easily generates negligible values. Exponential increase of the pressure as function of the gap provides reasonable behavior in terms of penetrations for large gap variations over a surface. Convergence can become difficult to obtain for very fast varying contact conditions since penetration overestimation can lead to numerically infinite forces.
This law has two parameters, p_z in Pa being the pressure at zero gap (curve offsetting as function of the gap), and λ in m^{-1} being the scaling factor (curvature of the exponential).
The contact stiffness for tangent contact states at contact integration point i is thus

$$k_{ci}(g_{ni}) = \lambda p_z e^{-\lambda g_{ni}} \tag{3.38}$$

eq*

The tangent contact stiffness varies with the gap at each points, which allows fine observation of system sensitivities to contact states in the modal domain.
As the definition of λ can be difficult, the exponential law can be alternatively set with parameter p_e instead of λ. This parameter stands of expected pressure at structural stiffness, and represents a maximal working target pressure for the model at which one will expect the exponential law to generate a stiffness density higher than a stiff linear coupling. The estimated stiffness is the same than for the bilinear law with `Kc=-1`, and λ is resolved as $\lambda = \frac{K_c}{p_e}$. Parameter lambda is then set in the law after initialization.
Using an exponential law can lead to very high stiffness densities, that for larger gaps may lead to very high values that would compromise numerical conditioning. It is thus possible to define a stiffness bound that will saturate the exponential growth after a critical gap value. One then defines parameter `KcMax` as the maximum stiffness density allowed to the exponential formulation. An associated gap $g_{max} = \frac{-log(KcMax/p_z\lambda)}{\lambda}$ defines the threshold after which the exponential law is bounded so that for $g_{ni} > g_{max}$

$$\begin{array}{l} p(x_i) = -K_{c_{max}}\left(g_{ni} - g_{max}\right) + p_z e^{-\lambda g_{max}} \\ k_{ci}(x_i) = K_{c_{max}} \end{array} \tag{3.39}$$ eq*

p_z, of input name pz, λ of input name lambda, and $K_{c_{max}}$ can be set to *ValP*, *ValL* and *ValKM* in Fu as a string of format *'Kc ValP* lambda *ValL'* or as a line vector *[ValP ValL ValKM]*. The string format only accepts defining parameter p_e of input name pe instead of lambda. Note that pe and lambda are mutually exclusive parameters, with the priority set on lambda.

_laws_tab

Tabulated contact law, 3 : tabular

Tabulated contact laws are user defined or built using nlutil('@Ctc'). Using chandle both linear an piecewise cubic interpolation are supported. The curves are referenced using a curve identifier (see sdtweb curve for more details). The table is a gap pressure relationship, with field X being the gap and field Y being the pressure.
For linear interpolation, a contact point i the output pressure is interpolated between the two gap values bounding the exact gap, noting $\mathcal{G}$ the set of provided gap values in the table

$$\left\{ \begin{array}{l} p(x_i) = (1-r_i)p_{1i} + r_i p_{2i} \\ g_{1i} = \max_{g\in\mathcal{G}} g \leq g_i \\ g_{2i} = \min_{g\in\mathcal{G}} g > g_i \\ r_i = \frac{g_i - g_{1i}}{g_{2i} - g_{1i}} \end{array} \right. \tag{3.40}$$ eq*

If g_i is outside the table bounds, extrapolation is performed following the slopes of the first or last table segment, such that if g_i is lower than the lower bound g_{1i} and g_{2i} are the first two values of the table, and if g_i is higher than the higher bound g_{1i} and g_{2i} are the last two values of the table.
The contact stiffness for tangent contact states at contact integration point i is taken as the slope of the segment in which g_i is found

$$\left\{ \begin{array}{l} k_{ci} g_i = \frac{p_{2i} - p_{1i}}{g_{2i} - g_{1i}} \\ g_{1i} = \max_{g\in\mathcal{G}} g \leq g_i \\ g_{2i} = \min_{g\in\mathcal{G}} g > g_i \end{array} \right. \tag{3.41}$$ eq*

In case of extrapolation, the slopes of the first or last segments are taken.
The curve must be stacked in the model with a proper .ID field. The curve of ID *ID* can be assigned to Fu as a string of format *'CurveID ID'* or as a a scalar *ID*.
xxx give reference to cube example

_laws_pow

Power contact law, 4 : tabular

Power laws can be used in the same way as exponential laws, for which it can be easier to fit parameters in some applications.
This law has a list of parameters k_{pm} defining a stiffness density associated to power m as

$$p(x_i) = max\left(0, \sum_m k_{pm}(-g_{ni})^m\right) \tag{3.42}$$ eq*

The associated contact stiffness is

$$k_{ci}(g(x_i) < 0) = \sum_m m k_{pm}(-g_{ni})^{m-1}) k_{ci}(g(x_i) >= 0) = 0 \tag{3.43}$$ eq*

k_{pm}, of input name kpm, can be set to *ValM* as a string format kpm *ValM* or as a 2 colum matrix *[m ValM;...]*.

_laws_cdyn

Exact Coulomb law, 1, 11 : coulomb strict

The Coulomb law is here expressed dynamically as function of the sliding direction (unidirectional for 1, bidirectional for 11), the friction constraint is a two direction vector at contact point i, expressed as function of the sliding velocity w (see (3.25)) as

$$\mathbf{f}_t(x_i) = -\frac{\mu p(x_i)}{\|\mathbf{w}\|}\mathbf{w} \tag{3.44}$$ eq*

This law takes in input the friction coefficient μ (input name Mu), and optionally the tangent sticking stiffness parameter κ (input name kappa).
.Fv='Mu *ValM* kappa *ValK*' was used prior to chandle. .Fu='n3s13{Kt1e12,Mu.1,kappa.2}' is the new format.

_laws_creg

Regularized Coulomb law with linear slope, 2, 12 : coulomb reg

The Coulomb law is here expressed dynamically as function of the sliding direction, the friction constraint is a two direction vector at contact point i, expressed as function of the sliding velocity w as

$$\mathbf{f}_t = -\min\left(c_t\|\mathbf{w}\|, \mu p\right)\frac{\mathbf{w}}{\|\mathbf{w}\|} = -\min\left(c_t, \mu p(x_i)/\|\mathbf{w}\|\right)\mathbf{w} \tag{3.45}$$ eq*

The slope is then to the friction force between zero sliding velocity and $\|\mathbf{w_i}\|_2 = \mu p(x_i)/k_t$. The saturation threshold thus depends on the level of contact pressure at each contact point.
This law takes in input the friction coefficient μ (input name Mu), the regularization slope k_t (input name CtLin), and optionally the tangent sticking stiffness parameter κ (input name kappa).
μ, of input name Mu, c_t of input name CtLin, and κ of input name kappa, can be set to *ValM*, *ValCT* and *ValK* in Fv as a string of format 'Mu *ValM* CtLin *ValCT* kappa *ValK*' or as a line vector *[ValM ValCT ValK]*.

_laws_carc

Regularized Coulomb law with arctangent, 3, 13 : coulomb arctan

The Coulomb law is here expressed dynamically as function of the sliding direction, the friction constraint is a two direction vector at contact point i, expressed as function of the sliding velocity w as

$$\mathbf{f}_{ti} = -\frac{2\mu p(x_i)}{\pi}\text{arctg}\left(c_t\|\mathbf{w_i}\|_2\right)\frac{\mathbf{w_i}}{\|\mathbf{w_i}\|_2} \tag{3.46}$$

eq*

The arctangent shape is here exploited to approximate the friction law, the advantage being that this function is infinitely continuously derivable thus alleviating potential threshold effects.
This law takes in input the friction coefficient μ, the regularization slope c_t, and optionally the tangent sticking stiffness parameter κ.
μ, of input name `Mu`, c_t of input name `CtLin`, and κ of input name `kappa`, can be set to *`ValM`*, *`ValCT`* and *`ValK`* in `Fv` as a string of format `'Mu` *`ValM`* `KtLin` *`ValCT`* `kappa` *`ValK`*`'` or as a line vector *`[ValM ValCT ValK]`*.

_laws_crsl

Regularized Coulomb law with scaled linear slope, 4, 14 : coulomb scaledreg

The Coulomb law is here expressed dynamically as function of the sliding direction, the friction constraint is a two direction vector at contact point i, expressed as function of the sliding velocity w as

$$\mathbf{f}_{ti} = -\min\left(c_t\|\mathbf{w_i}\|_2, \mu\right)p(x_i)\frac{\mathbf{w_i}}{\|\mathbf{w_i}\|_2} \tag{3.47}$$

eq*

The slope is then to the friction force between zero sliding velocity and $\|\mathbf{w_i}\|_2 = \mu/k_t$. The saturation threshold is thus independent on the level of contact pressure at each contact point.
This law takes in input the friction coefficient μ, the regularization slope c_t, and optionally the tangent sticking stiffness parameter κ.
μ, of input name `Mu`, k_t of input name `CtLin`, and κ of input name `kappa`, can be set to *`ValM`*, *`ValCT`* and *`ValK`* in `Fv` as a string of format `'Mu` *`ValM`* `CtLin` *`ValCT`* `kappa` *`ValK`*`'` or as a line vector *`[ValM ValCT ValK]`*.

s*hbmfun

Function reference

Contents

List of the SDT-HBM module functions	
Function	Description
`d_hbm`	Open source HBM examples
`d_tdoe`	Open source time DOE examples
`hbm_solve`	Solving tools
`hbm_post`	Post process tools
`hbmui`	User interface
`hbm_utils`	Maintenance

d_hbm

Purpose

Open source examples for HBM

Syntax

```
d_hbm
```

`TestBeamNL`

Simple bending beam with localized non-linearity.

```
model=d_hbm('testBoltedJoint')
```

d_tdoe

Purpose

Open source examples for time experiments.

Syntax

```
d_tdoe % opens tag list of tutorials
```

`MeshCfg URN definition of meshes`

The handling of mesh configuration is done by `d_mesh MeshCfg`. The convention is that the mesh URN is of the form `Mesh:Case:NL`. Thus `d_hbm(0D):DofSet:0Dm1t` says that

- the `d_hbm('Mesh0D')` command is called to obtain the mesh
- the `d_hbm('CaseDofSet')` command is called to set case information (loads, sensors, ...)
- the `d_hbm('NL0Dm1t')` command is called to set non-linearity information. Here `0Dm1t` refers to a specific material.

`SimuCfg URN definition of time experiment`

The handling of time simulation configuration is done by `d_fetime SimuCfg`. Different conventions are being developed for different types of experiments.
`SteppedSine{.5,1,10}:C0{0,15}:C1{2.5,10}` is used to characterize stepped sine tests. The URN is split as follows

- `SteppedSine{.5,1,10}` gives the target frequencies in Hz.
- `C0{0,15}` gives the harmonic 0 or static offset. xxx scaling convention
- `C1{2.5,10}` gives the first harmonic or time signal of the form $cos(\omega t)$
- xxx 'NperPer',2e3,'Nper',1,'iteStab',20

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 mentioned 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.

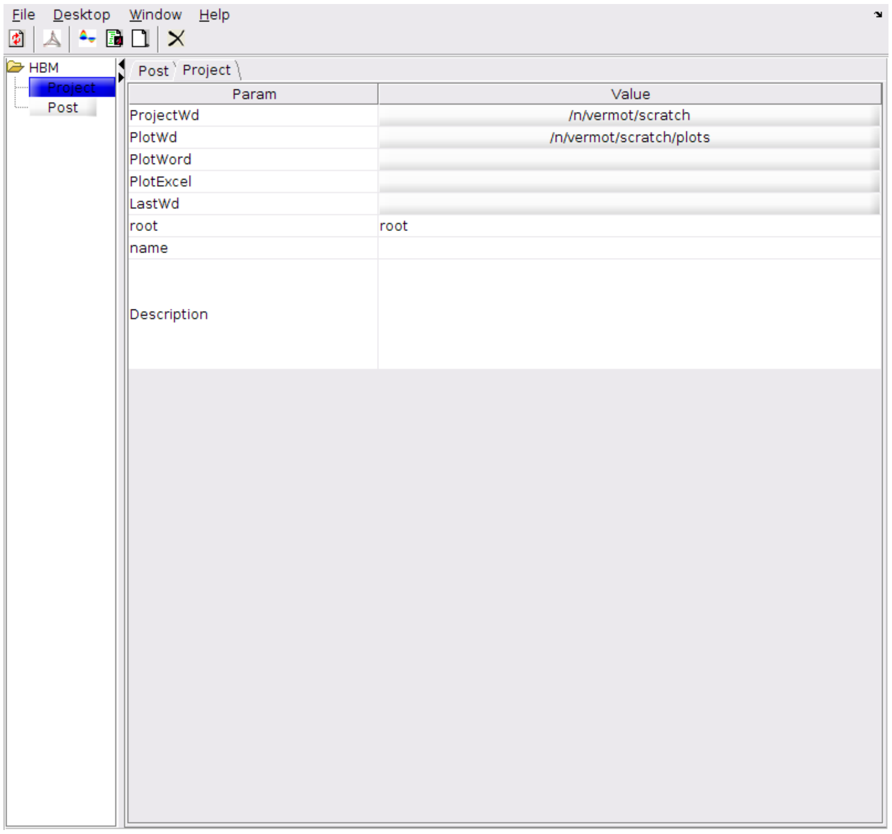
Command `PARAMUI` returns the complete application UI structure.

`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'));`

Tabs

Project

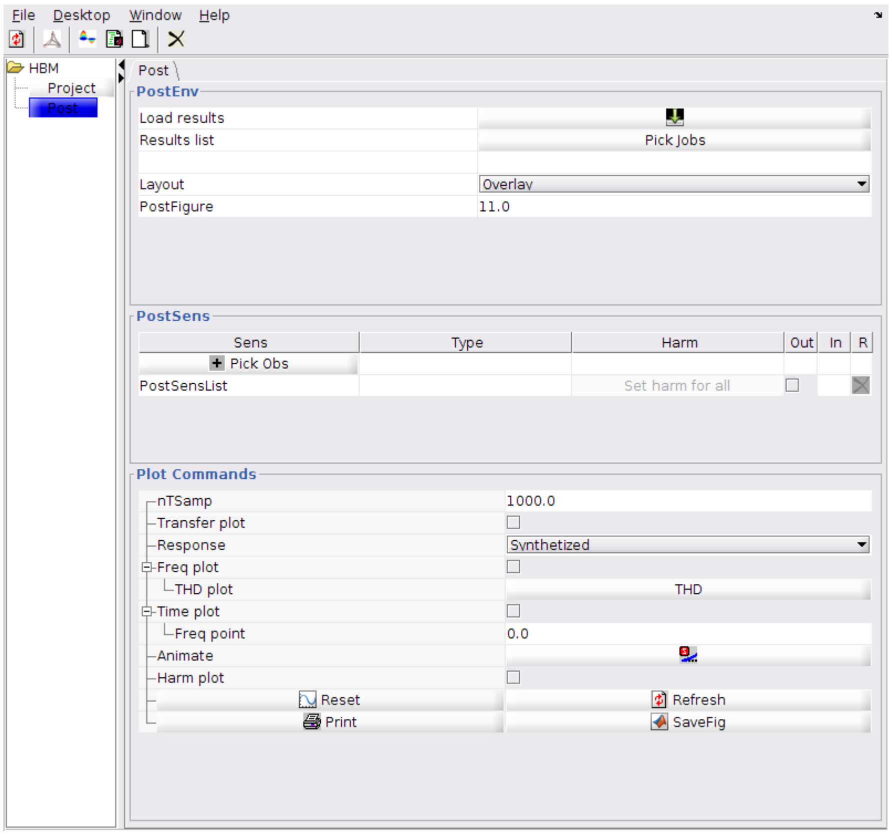


Param	Value
ProjectWd	/n/vermot/scratch
PlotWd	/n/vermot/scratch/plots
PlotWord	
PlotExcel	
LastWd	
root	root
name	
Description	

Figure 4.1: Project tab in initial state `f:UI_Tab_Project_0`

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 4.2,



Load results	
Results list	Pick Jobs
Layout	Overlay
PostFigure	11.0

Sens	Type	Harm	Out	In	R
Pick Obs					
PostSensList		Set harm for all	☐		

nTSamp	1000.0
Transfer plot	☐
Response	Synthetized
Freq plot	☐
THD plot	THD
Time plot	☐
Freq point	0.0
Animate	
Harm plot	☐
Reset	Refresh
Print	SaveFig

Figure 4.2: Post tab in initial state f:UI_Tab_Post_0

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.
- 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`*)

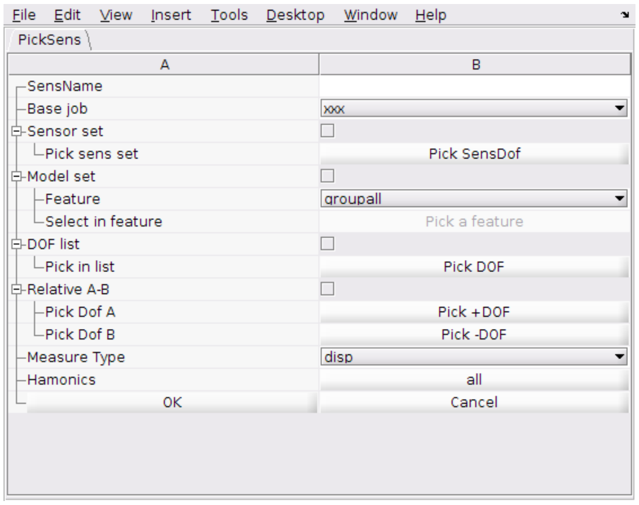
  - `Post.RespSyn` The type of synthesis to be performed, either `Synthesized` to synthesize the response on observation with all retained harmonics, `byHarm` to generate an observation response per harmonic, or `byShape` to generate an observation response per time shape.
  - `Post.nTSamp` Provides the 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.
  - `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 refresh the display with the current setup.
  - `Post.PlotPrint` To print the the display into a plot, support for automatic reporting is provided through the `Porject` tab setup.
  - `Post.PlotSaveFig` To save a figure containing the current display.
  - `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 4.3 opens.



A	B
SensName	
Base job	xxx
Sensor set	☐
Pick sens set	Pick SensDof
Model set	☐
Feature	groupall
Select in feature	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	disp
Hamonics	all
OK	Cancel

Figure 4.3: 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)`.

  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.
  - `-q0inTR` 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.

- 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.

- 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` synthesizes and outputs a transient NL trajectory (in the standard `fe_time`NL 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

  – `nTSamp`*`val`* 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.

  – `coef`*`val`* to apply an amplitude coefficient *`val`* to the state.

  – `NeelUNL`*`val`* 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 application state can be recovered using XF=hbm_post('init').
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 synthesize 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 synthesized for the current object state. The other methods only return the field corresponding to their name.

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

`AFMap`

Performs amplitude/frequency response MAPS, possibly in a given subspace.
`Z=hbm_solve('AFMap',model,RO);`
`model` is a standard SDT-HBM model. 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 (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.
- `-itInit`*`stra`* 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 amplitude, 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 initialization and proper load definitions.

`fe_time[,cleanup]`

Command `fe_time` is called back by `fe_time`as 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 display 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.
- `-cf`*`val`* 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.

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 intersecting 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');cf=feplot;model.DOF=cf.def.DOF;
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)
```

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);`
`Z0` is the initial state provided to the solver, `opt` is the internal solver structure, `j1` is an interaction 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 predicted 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.

@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 of (1.118). `opt.N` and `opt.harm` (representing k/ν) are mandatory fields.

@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-evaluated to provide a linearly updated trajectory.

InitHBM

The subfunction @intHBM performs HBM solver specific initialization based on a SDT model with non-linearities. [model,Case,opt,Z0,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, Z0 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 iterative 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 corresponding 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, ki the current Jacobian, and 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.

@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 differentiated from `opt.w` to allow customized 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 `r` the harmonic residue, and `ki` 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 obtaind 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 utilities.

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 address in your Matlab preferences (File/Preferences/

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
```

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', ...)
```

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:

- `.C`*i* 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.

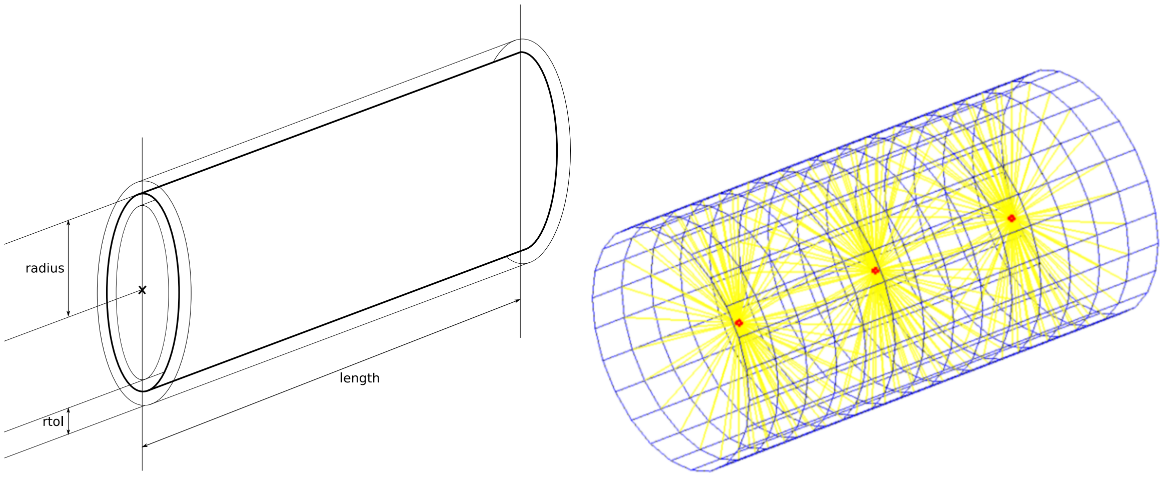


Figure 4.4: ConnectionCyl

fig

InitV

q0=nl_spring('InitV',model,d0,RO);
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,RO]=nl_spring('InitV-optim',model,d0); can be used to find input parameters RO. 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 also t_nlspring('2beam') example.

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. This is the low level implementation of calls documented in nl_solve TgtMdl. The said Jacobian must take non-linearities into account and is thus of the form

$$k_j = [b]\left[\frac{\partial s_{nl}}{\partial u_{nl}}\right][c] \tag{4.1}$$

eq*

the output is controlled by the value of NL.Jacobian.

- 0 gives no Jacobian.
- 1 use finite differences to evaluate Jacobian.
- 2 fixed Jacobian.

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:
  - freq 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: $\phi_j^T[K_{NLlink}]\phi_j$.
- `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.
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=fe_simul('Follow`*i*`',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=fe_simul('Follow1',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 .FnlIiplot 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.FnlIiplot=1; % define FNL tracker
model=stack_set(model,'info','OutputOptions',r1);
opt=stack_get(model,'info','TimeOpt','GetData');
def=fe_time(opt,model);
```

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-gamm .51 to introduce numerical damping by directly giving gamma.
- TimeOptStat to perform static simulations (see also fe_time nl_solve).
- TimeOptTheta to perform time simulations with the θ-method (see fe_time). Numerical damping can be introduced using TimeOptTheta-alpha .05, the specified α value will be added to θ, 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.

- dt*val* time step.
- ts*N* number of time steps.
- tend*val* optional end time
- tInit*val* initial time.
- AlphaR*val* a global Rayleigh damping mass coefficient (applied to the model total mass).
- BetaR*val* a global Rayleigh damping stiffness coefficient (applied to the model total stiffness).
- maxNout*N* requests an output subsampling strategy such that only *N* times equally spread over the simulation time span are output.
- RelTol*val* requests a specific relative tolerance for the convergence of iterative schemes.
- -gamma*val* requests a specific γ coefficient (default to .5) of the Newmark scheme. For the non explicit versions, β is adapted to ensure unconditional stability of the scheme.
- -theta*val* requests a specific θ coefficient (default to .5) of the Theta method.
- -acall*str* provides a series of command options applied to the AssembleCall generation.
- -fclean*str* provides a series of command options applied to the FinalCleanupFcn generation.
- -jcall*model* 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_solve('TimeOpt dt1e-6 ts3e5 maxNout1e4 -acall"lumpedMass"');
% call with struct input
RO=struct('dt',1e-6,'ts',3e5,'maxNout',1e4,...
'acall','lumpedMass');
opt = nl_solve('TimeOptExplicit',RO);
% basic call with model input
model = nl_solve('TimeOpt dt1e-6 ts3e5 maxNout1e4 -acall"lumpedMass"',[],model);
% call with struct and model input
model = nl_solve('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.

  `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

- -BlockSave*N* 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.

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.
- .FnlIiplot 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);
```

nl_spring

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 + b*(snl_fun) + fnl_int -fc;
```

In this formula, `b*(snl_fun)` refers to the legacy m file implementations (non `chandle`) which expect the `nl_` function to fill a `unl` vector multiplying `b` (`.snl` field support was not activated). This was incorrectly noted `-fnl` in earlier documentation the minus sign coming from the convention of considering non-linear forces as external rather than internal (the convention used from now on). `fnl_int` refers to `chandle` implementations that match classical conventions : traction leading to positive stress in continuous mechanics, positive over-closure (negative gap) leading to positive pressures in contact mechanics.
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 $T_l^T M\Phi \left[\backslash 2\zeta_j\omega_{j\backslash} \right] \Phi^T M T_R$ 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.

chandle

`chandle` objects are used to streamline communication between mex and MATLAB in iterative processes. They are used in various `nl_solve` calls and in particular for `ModalNemwark` and `ExpNewmark`.

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 vhandle.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 (explict or implicit).

ExpNewmark

ExpNewmark is an implementation of the Newmark scheme when assuming a fixed diagonal mass matrix for large explicit dynamic problems.

nl_inout

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]`
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  {
   int    Nk,Nnl;
   double RayleighM,RayleighK,tc;
   double *u,*v,*a,*FNL;
  };
// Default function handle
typedef void(*__Fu)(chandle* ph, struct _ROr ROr);
```

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 associated with elements supporting the kinematics (`cbush` for point connections, see section 1.2.1 , zero thickness volumes for surface connections (where two layers of coincident nodes are considered for a `hexa8` or `penta6` element, see section 1.2.2), volume elements for 3D applications (see section 1.2.3) or the deprecated observation/sensor command/loads as detailed in section 1.2.5 .
The general form of the non-linearity $f_{NL} = b \times f(C.u, C.v)$ is detailed in section 1.1 .
For a list of implemented non-linear constitutive laws xxx
The `pro.NLdata` structure has fields described in section 1.6.1 (with the need to distinguish the form for model declaration and during time integration).
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_contact

Supports non conform fixed matching contact (squeal applications for example) and (surface contact large displacement, as in rail/wheel interaction for example). For conform meshes, zero thickness elements associated with `p_zt` can be used.
See `p_contact`, `ctc_utils`.

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 ver09 [29].
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_inout

Purpose

The generic form of NLdata specification is discussed in section 1.6 .

DofSet

Implementation of linear or large rotation setting of DOF values from time curves of a fe_case DofSet entry. sdtweb('_eval;','d_fetime.m#NLNewmark_LrDofSet').
Supported variants are

- large rotation trajectories (using MBBryan) uses .unl(:,:,1) corresponding to large displacement and updates .unl(:,:,j1) when changing time step to allow velocity computations (currently only valid for fixed time step). Requires setting .KeepDof=1 and enforcing 6 DOF associated with the master node.
- translation trajectories (xxx).

Power

uses NLdata.opt=[comstr('Power',-32) k n] and $s_{nl} = k\ u_{nl}^n$.

FuTable

uses tabular definitions in either NLdata.Fu or NLdata.Fv. If you want a case with both simultaneously to be re-implemented please provide a test case. Implements a Jacobian using tangent stiffness. Run example with sdtweb('_eval;','d_fetime.m#ModalNew_FuTable').

K_t

Implement time varying matrices and DofSet xxx.

MexIOa

Implements callback to user defined .m file implementations of non-linearities as detailed in section 1.3.5 .

SCLd

Large displacement surface contact supported as part of the contact module (see sdtweb(xxx)). Possible application : rail/wheel contact. This supports low pass filtering of contact forces, using an evolution equation of the form $\dot{F}_c/\omega_c + F_c = k_c g$ (for a positive constant) or $\dot{F}_c/\omega_c + F_c = k_c\sqrt{(}g)$ for a negative constant. This avoids incorrect bouncing of stiff contacts in implicit computations and ω_c should be a fraction of the sampling frequency (inverse of time step) or the maximum mesh frequency.

`LRFu`

Large rotation tabular spring for multi-body applications. `cbush` kinematics are expected and rotations are assumed using `Bryan` angles in radians. Requires one non-linearity for each spring (not currently vectorized). The `NL.Node` field gives slave and master node for each body (4 nodes). The observed motion is given at the master node and large rotation is used to determine the current position of the slave node. With single axis springs (large rotation rod) the position of the two slave nodes should be disjoint.
The stress vector used by this non-linearity contains 24 elements

- `sx1,sy1,sz1,srx1,sry1,srz1` forces and moments on node 1
- `sx,sy,sz,srx,sry,srz` forces and moments on node 2
- `PX1,PY1,PZ1,rx,ry,rz` absolute position of node 1 and rotations
- `ux,uy,uz` relative displacements in x,y,z directions, `sex,sey,sez` force in element frame.

`slab sensor non-linearity`

This non-linearity supports observation of various quantities during FEM computations. The base definition is associated with the `sdtweb('sensor# scell')`

- For direct resultant sensors in time integration schemes using enforced displacement `-fieldOut4`, leading to `NL.iopt(6+isens)==4`. See more details in xxx

Needs documentation, see `sdtweb d_fetime('slab')`.

```
NL=struct('type','nl_inout','slab',{li(:)},'UserObs',dyn_solve('@doObs'));
mt=stack_set(mt,'pro','Sensors',struct('il',1e5-1,'type','p_null', ...
     'NLdata',NL));
```

Note that in predictor/corrector schemes, it may be necessary to recompute the residual to obtain the correct residual.

`temp still undocumented`

- `FuExpon` $s_{nl} = a\ b^{u_{nl}}$.
- `uMaxw` generic C implementation of `nl_maxwell.m` file

- FuDahlC Dahl model with constant force
- STS PSA scalar STS
- CLIMA2 connector non-linearity developed with Marco Rosatello
- FuFric basic friction model with $F_k = Ku$ bounded by $\pm F_{max}$.
- nl_bset xxx
- DofSet xxx

Non linearities list (deprecated)

Purpose

`nl_maxwell (deprecated)`

`nl_maxwell` describes rheological models using stiffness and damping. *Deprecated implementation that should now be called with an* `nl_inout`.

`.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, should be initialized from `StoreType`.
`Ncell`	number of cells.

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 4.5.

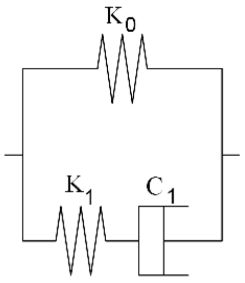


Figure 4.5: Standard viscoelastic model. f:zenermdl

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

$$F(s) = K(s)X(s) = \frac{K_0K_1 + (K_0+K_1))C_1 s}{K_1 + C_1 s} = K_0 \frac{1+s/z}{1+s/p} \tag{4.2}$$ eq*

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

$$p = \frac{K_1}{C_1} \tag{4.3}$$ eq*

$$z = \frac{K_0K_1}{(K_0+K_1)\,C_1} \tag{4.4}$$ eq*

The maximum loss factor is

$$\eta_m = \frac{p-z}{2\sqrt{pz}} = \frac{1}{2}\frac{K_1}{\sqrt{K_0\,(K_0+K_1)}} \tag{4.5}$$ eq*

and obtained for pulsation

$$\omega_m = \sqrt{pz} = \frac{K_1}{C_1}\sqrt{\frac{K_0}{K_0+K_1}} \tag{4.6}$$ eq*

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 4.6.

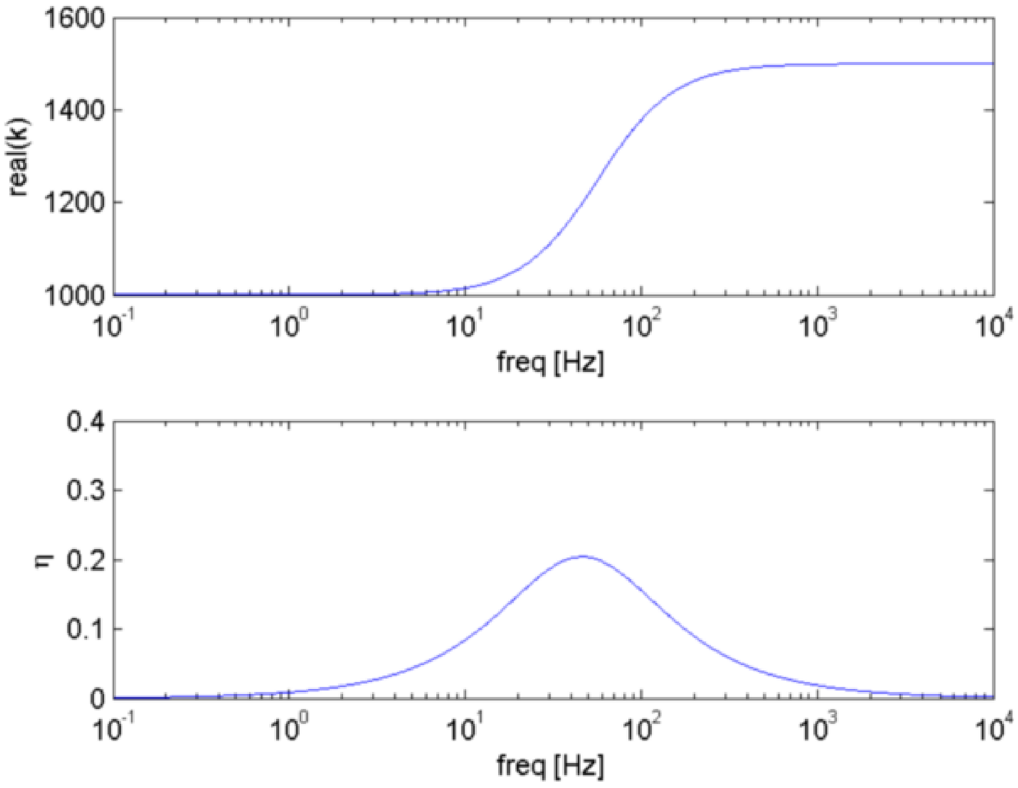


Figure 4.6: Module and loss factor. f:zenerexgeta

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)    ]);
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
```

Non linearities list (deprecated)

DofKuva

DofKuva defines a non linear load of the form
$av^e_{Dof}\,[K]\,\{V\}$ 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) $(u)^{exponent}.V$ NDdata fields for this non-linearity are

.type	'DofV'
.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.
.exponent	Exponent of the DOF.
.def	data structure with fields .def which defines vector V and .DOF which defines corresponding DOF.

nl_spring

nl_spring defines a non linear load 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.
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] bumpstop. 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:

```
RT=struct('nmap', vhandle.nmap);
% sdtweb d_fetime Mesh2DOF        % For meshing scrip
% sdtweb d_fetime LegacyBumpStop  % For NLdata definition
li={'MeshCfg{d_fetime(2DOF),LegacyBumpStop{Z0}}'
      'SimuCfg{Imp{1m,10,uva110}}'; % implicit NLNewmark simulation
      'RunCfg{Time}'};disp(comstr(li(1:2:end)',-30))
RT.nmap('CurExp')=li; sdtm.range(RT);
model=RT.nmap('CurModel');def=RT.nmap('CurTime');

li{3}='SimuCfg{Exp{1m,10,uva110}}'; RT.nmap('CurExp')=li;% Same using explicit
sdtm.range(RT);mo2=RT.nmap('CurModel');d2=RT.nmap('CurTime');

figure(10);clf;% plot some comparison between results
subplot(211);plot(def.data,[def.def' d2.def']);xlabel('Time [s]');ylabel('displacement'
subplot(212);plot(def.data,[def.v' d2.v']);xlabel('Time [s]');ylabel('velocity')
legend('Implicit','Explicit');setlines;
sdth.os(10,'@OsDic',{'ImGrid','ImSw80','ImTight'})
```

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

```
% define model:
L=1; b=1e-2; h=2e-2; e=1e-3; % dimensions
model=[];
model.Node=[1  0 0 0  0 0 0; 2  0 0 0  L 0 0];
model.Elt=[Inf abs('celas') 0 0;
           2 0  2 0     100  1 110 0;   % linear celas
           Inf abs('beam1') 0 0;
           1 2  1 1    0 1 0 0
           ];
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','nl_inout',...
                              '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 θ_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

$$\begin{array}{rcl} \{x_A\}_G & = & [R_{GL}]\left(\{p_A\}+\{u_A\}_L\right) \\ \{x_B\}_G & = & \left(\{p_B\}+\{u_B\}_G\right) \end{array} \tag{4.7}$$

eq*

which leads to expressions of the loads as

$$\begin{array}{rcl} \{F_A\}_L & = & [R_{LG}]\left(K\left(\{x_B\}_G-\{x_A\}_G\right)\right) \\ \{F_B\}_G & = & K\left(\{x_A\}_G-\{x_B\}_G\right) \end{array} \tag{4.8}$$

eq*

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

$$\begin{array}{rcl} \{\dot{x}_A\}_G & = & [R_{GL}]\left(\{\dot{u}_A\}_L+\{\omega(t)\}\wedge\{p_A+u_A\}_L\right) \\ \{\dot{x}_B\}_G & = & \{\dot{u}_B\}_G \end{array} \tag{4.9}$$

eq*

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

$$\begin{array}{rcl} \{FC_A\}_L & = & [R_{LG}]\left(K\left(\{\dot{x}_B\}_G-\{\dot{x}_A\}_G\right)\right) \\ \{FC_B\}_G & = & K\left(\{\dot{x}_A\}_G-\{\dot{x}_B\}_G\right) \end{array} \tag{4.10}$$

eq*

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

$$\left\{\begin{array}{c} q_{AG} \\ q_{BG} \end{array}\right\} = \left[\begin{array}{cc} R_{GL} & 0 \\ 0 & I \end{array}\right]\left\{\begin{array}{c} q_{AL} \\ q_{BG} \end{array}\right\} \tag{4.11}$$

eq*

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

$$k\left[\begin{array}{cc} I & -I \\ -I & I \end{array}\right] \tag{4.12}$$

eq*

which leads to the following stiffness matrix

$$\left[\begin{array}{cc} R_{GL}^T & 0 \\ 0 & I \end{array}\right] k \left[\begin{array}{cc} I & -I \\ -I & I \end{array}\right]\left[\begin{array}{cc} R_{GL} & 0 \\ 0 & I \end{array}\right] = k\left[\begin{array}{cc} I & -R_{GL}^T \\ -R_{GL} & I \end{array}\right] \tag{4.13}$$

eq*

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 θ_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
- .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 (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.
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 xb 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);
```

rod1

The rod1 non-linear connection is a simple penalized rigid link. One considers two nodes A and B (see figure 4.7).

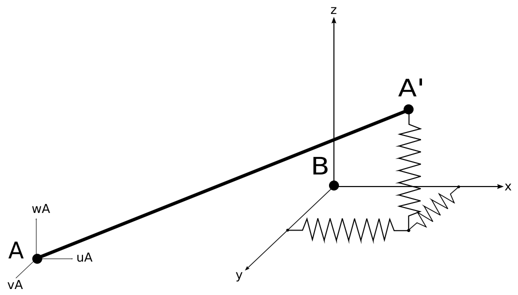


Figure 4.7: Large rotation rod functional representation. f:rod1fig

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.
The global non linear load associated with the rod is thus

$$F_{rod} = k_r \left(\|\{x_B - x_A\}\| - L_0\right) \frac{\{x_B-x_A\}}{\|\{x_B-x_A\}\|} \quad (4.14)$$ eq*

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

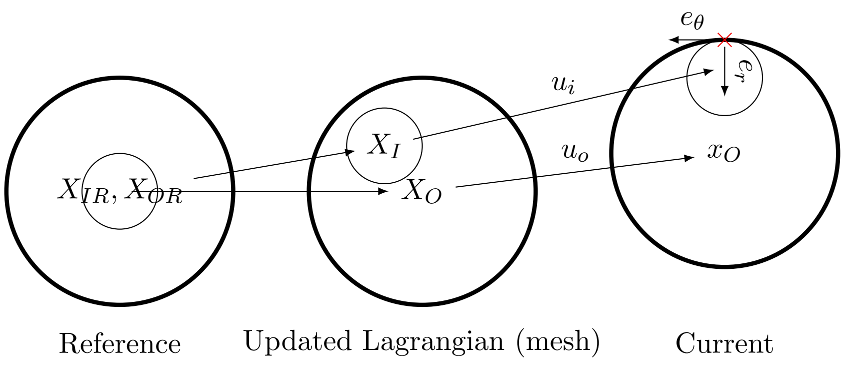
$$\{q_{A'}\} = \begin{bmatrix} I & \left[\vec{AB}\wedge\right] \\ 0 & I \end{bmatrix} \{q_A\} \tag{4.15}$$

eq*

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

$$u_{NL} = x_O - x_I = u_O - u_I + (X_O - X_I) = [c]\{q\} + u_{NL0} \tag{4.16}$$ eq*

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_θ 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

$$g = \{e_r\}^T \{u_{NL}\} - (R_O - R_I) \tag{4.17}$$ eq*

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

$$\{f_I\} = \{-f_O\} = f(g)\{e_r\} \tag{4.18}$$ eq*

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.
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.

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 non-linearities 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 non-linearity 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 EnerP and EnerK 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.

Creating a new non linearity: `nl_fun.m`

```
% A standard SDT model
model=struct('Node',[1  0 0 0  0 0 0; 2  0 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_simul`for 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 `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_eig`for real modes and `fe_ceig`for 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 detailing 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_time`simulations 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_time`for the preliminary state computation. With `-stat` option (default) the stack entry `info,`

`TimeOptStat` will be sought 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 sought and used if found, if not the result will use the `-skip` mode. In `-time` mode, the model stack entry `curve,TSIM` will be sought 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 `iiplot`format.
- `-fullDOF` to output the deformation fields restituted on the unconstrained DOF.
- `-keepT`*`val`* (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 `feplot`on-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_eig`format.
- `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 syntax. 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 `fe_eig`or 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 mo
  ```

- 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_solve('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

- `-cf`*`i`* 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.
- `-globalMaxTol`*`val`* 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)'});
```

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. The low level implementations are documented in nl_spring NLJacobianUpdate (for example keepLin interaction are documented there).

- 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.
  - SEPro to initialize p_superentries for the generated superelements.
  - setPar to setup parameters associated to the generated superelements, conforming to the content of the nlpro.
  - keepNL*ProId* not to generate tangent superelements for the non-linear properties specified by their *ProId*.
  - -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^{npar} 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), q0cbk is a callback in cell-array format.

staticInterp2 implements a more generic method using vhandle.matrix NLJac types for richer interpolation. In such case one can provide a field .Range in the input options to force a richer DoE for the static basis. By default griddedInterpolant is used so that the given Range must be compatible with the interpolation strategy.

```
% 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 non discretized 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 lc*val* allows specifying a characteristic length for Gmsh.
- Command option lcmin*val* 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 `group`*`val`* is used in combination to the `s`ingle 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.

- `lc`*`val`* 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 avaiable

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 `lsutil`to 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.

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 radius and washer (or rivet head radius). The mesh generated between both radius 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*v*al allows setting the modified mesh MatId to *v*al.
- Command option ProId*v*al allows setting the modified mesh ProId to *v*al.
- 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 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
RO=struct('Orig',[ 3 1 0;6 1 0;9 1 0],...
 'radHole',[.2;.2;.2],...
 'radWash',[.8;.8;.8]);

model=nl_mesh('Rivet',model,RO);
cf=feplot(model);
```

`ShellSkin`

Generation of a skin mesh from shell elements.
Syntax: `mo1=nl_mesh('ShellSkin,model,RO`. `model` is a model with shell elements, `RO` is a facultative option structure input. Output `mo1` is a model with added volume elements.
By default, all shell elements are selected, and the average thickness is computed using the associated integration rules. A symmetric extrusion following the shell normals is performed.
Command option Get outputs an exteral model rahter than an addition to the existing model. The following options are available:

- .sel to provide a Findelt command to select target shell elements.
- .twoSided to generate a volume extrusion from both sides.
- .vol to define the volume mesh format: 0 will generate shell skin meshes linked with rigid elements, 1 will generate volume elements, 2 will generate volume elements linked with rigid elements to the shell.
- .MatId to assign a new material identifier to the new elements. Set to NaN to leave initial MatId.
- .ProId to assign a specific ProId to the new elements, one can set to 0 if only the topology is used.
- .New to add the new elements to the existing model, with new EltId.

```
% Generate a shell mesh
model=femesh('testquad4 -divide 4 4');
% Generate a wolume mesh with rigid connections to the shell nodes.
RA=struct('sel','groupall','vol',2,'twoSided',1);
mo1=nl_mesh('ShellSkinGet',model,RA);
cf=feplot(mo1); fecom(cf,'colordagagroup')
```

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,RO,ofact(1,'lu'));`

`ki` is the matrix that is used for the optimization. `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

Following example optimize only solving:

```
ki=rand(20);
RO=struct('nCompt',100,... number of computation for result averaging
          'maxDomain',2.^[4:7],... parameter 1
          'maxZeros',logspace(-3,1,5),... parameter 2
          'refineStep',3); % refine results to most relevant parameters
spfmex_utils('ofactoptim solve-refine',ki,RO,ofact(1,'lu')); % method,solve,fact,-setop
```

nl_bset

Purpose

Non linearity to support handling of enforced displacement

This is implemented in `nl_spring('nl_bset')`. This currently assumes the existence of a stiffness and xxx viscous damping xxx.

ctc_utils

Purpose

High level contact handling in SDT/NL and SDT/contact module.

`Generate, GenerateContactPair`

This command generates contact data necessary to perform computation of observation and tangent matrices, and possible feplot displays (when using `p_contact` this command is handled with the syntax
`ctc_utils('Generate',model,data)`
`model` is a standard SDT model, `data` is a structure or a cell array of structures defining contact interactions. `data` has the fields

- `name` Optional. The interaction name.
- `master` an element selector string which result generates the Face set of the master surface, see `sdtweb FindElt`. The master is necessarily associated with elements in the model.
- `slave` An element selector string which result generates the Face set of the slave surface (for master/slave convention see section 3.2), see `sdtweb FindElt`.

The following fields are optional, and only required for tangent matrix computations. xxx why not set

- `contact` A string providing the contact law, or the corresponding integer, among the following list. `1`: linear; `2`: exponential; `4`: `tabular`.
- `Fu` A string providing the contact law parameters.
- `friction` A string providing the friction law or the corresponding identifier, among the following list. `0`: `none` ; `1` or `11`: `coulomb strict` ; `2` or `12`: `coulomb reg` ; `3` or `13`: `coulomb arctan` ; `4` or `14` `coulomb scaledreg`.
- `Fv` A string providing the friction law parameters.
- `ProId` to specify the target `ProId`.

If sequentially called for a single model, `Generate` will concatenate the contact data model with new data. Command option `-reset` resets the contact data and only keeps the result from the last call. For SDT/NL models,

- Command `GenerateContactPair` allows a direct generation of contact elements in the model.
- generating the minimal contact data model outside the global one for visualization optimization, command option `Obs` can be used.

Set SetPro

This command handles high level contact/friction laws assignments
`model=ctc_utils('set`*contId par val*`',model);`
contId is an identifier of the contact law to define. One can either use `ctcname`*name*, with *name* the contact interaction name (corresponding to the model stack entry), or `proid`*val* with *val* the contact elements `ProId` attached to the contact law.
One can then assign any property *par* to `val` regarding contact and friction data in the command string. Accessible parameters are `ContactLaw`, `FrictionLaw`, `Fu`, `Fv`, `TangStick`, `Euler`, and other fields relative to contact are supported.
The alternative command `setEdit` allows editing fields `Fu` and `Fv` by only specifying the varying parameters instead of redefining the full sub-property.
The following examples illustrate the syntax use for a model with contact property `my_ctc` and `ProId` `1001`:

```
% two equivalent calls to assign a linear contact property
% of stiffness 1e10 and fixed jacobian value to 1e9
model=ctc_utils('set ctcname"my_contact" ContactLaw"linear" Fu"Kc 1e10" KcLin 1e9',mode
model=ctc_utils('set ProId 1000  ContactLaw 1 Fu"Kc 1e10" KcLin 1e9',model);
% call to assign a friction law with tangent sticking property
model=ctc_utils('set ProId 1000 FrictionLaw11 Fv"mu 0.3 kappa 1e5" TangStick 1',model);
% two equivalent calls to modify kappa in the friction law prop Fv
model =ctc_utils('set ProId1000 Fv"mu 0.3 kappa 1e3"',model); % need to specify the ful
model=ctc_utils('setEdit ProId 1000 Fv"kappa 1e3"',model); % with edit no need to speci
```

Definition of customized local orientation MAPs is possible using further arguments,

```
MAP=struct('dir',{{0,1,0,'v3x','v3y','v3z'}},...
           'lab',{{'v1x','v1y','v1z','v3x','v3y','v3z'}});
model=ctc_utils('set',model,'MAP',MAP);
```

remove

This command allows removing contact elements and/or properties in the model, and the contact coupling superelement `tgtctc`.
`model=ctc_utils('Remove`*contId*`',model)`;
contId is an identifier of the contact law to define. One can either use `ctcname`*name*, with *name* the contact interaction name (corresponding to the model stack entry), or `proid`*val* with *val* the contact elements `ProId` attached to the contact law.
The following command options are available:

- `all` allows removing all contact elements at once.

- `SEinModel` allows removing the tangent contact superelement `tgtctc` generated by command `TangentMdlinModel`.
- `obs` allows removing the minimal contact observation data in an SDT/NL model generated with command `GenerateObs`.
- `-keepSE` allows keeping the coupling superelement `tgtctc` in the model.
- `-keepStackSE` removes the superelement declaration but keeps the stack entry `SE,tgtctc`.

`GetCPro`

This command recovers contact properties in the model. The base syntax is `[cproid,cil,cpro]=ctc_util`
`model` is an SDT model with contact elements. The outputs are `cproid` a column vector providing the `ProId` of contact elements in the model, `cil` an `ElProp` matrix of `p_contact`entries corresponding to `cproid` and `cpro` stack entries of type `pro` assocition to the contact properties.
The following command options are available

- `slide` to recover contact elements with sliding friction only (`Euler` property activated).
- `-zt` to also recover elements with `p_zt` properties.
- `-ztOnly` to only recover elements with `p_zt` properties.

`GetObs`

This command outputs data necessary to perform gap observations and generalized force projections for external computations, and `feplot` selections to display contact fields data.
`data=ctc_utils('GetObs',model)`
`data=ctc_utils('GetObs ProId`*i*`',model)` `data=ctc_utils('GetObs Name''`*val*`''',model)` `data=ctc`

`model` is a standard SDT model, on which command `Generate` was performed, or which contains contact elements.
Command option `Name`"*val*" allows specifying a contact interaction name for which observation data are necessary.
`data` is a cell array containing as many lines as declared interactions. The first colum contains the type, here fixed as `nl_contact`, the interaction names, the third one contains contact data, in a structure format with fields

- `DOF` The DOF vector on which observations are computed.
- `NORT` Matrix $[C_{NOR}]$. This is the observation matrix of normal gaps at the contact points.
- `TANT` Matrix $[C_{TAN}]$. This is the observation matrix of relative tangential displacements at the contact points. It is empty if no friction is asked.

- wjdet The ponderation product $J(x_i)\omega_i$ for each contact point.
- GaussCoor Coordinates of the contact points.
- adof DOF vector corresponding to all contact data in the model, ordered as GaussCoor field , with DOF .50 for contact pressure, .51 for friction constraint along the first local direction, .52 (when existing) for friction constraint along the second local direction.
- iNL Index vector of current contact data in vector adof
- Sel A feplot selection for visualization purposes

By default, the observation is based on the constrained DOF (Case.DOF) of the global model.

- Command option NoT To generate the observation on the full model DOF.
- Command option NoPlace To generate the observation on the contact model.
- Command option -NoSel Not to generate the feplot selection.
- Command option -selOnly allows outputting the visualization mesh only.
- It is possible to provide a deformation structure with field FNL as a third argument to adapt the observation to the deformation DOF.

GetFNL

This command allows computing a posteriori FNL field from a deformation field missing the non-linear data.

```
def=ctc_utils('GetFNL',model,def);
```

GetWeight

This command output in a cell array for each asked ProId, a struct with fields EltId and wjdet with a center node integration rule to provide the weight, or surface associated to each contact element.

```
data=ctc_utils('GetWeight',model);
```

ParInit

This command sets up parameter data in contact properties. They can be used to obtain a parametered tangent model with command ctc_utilsTangentMdl.

```
model=ctc_utils('ParInit',model,par);
```

model is a standard SDT model, on which command Generate was performed, or an SDT/NL model with contact elements. par is a parameter definition entry or a cell array of such entries. See

`.param` field description in `fe_range` to obtain more information about these entries, the following restrictions apply

- field `.info` is mandatory and must provide the parameter type and `ProId` using a formatted string as *`typ`*`-proid`*`val`*. Available types are
  - `Kn` to apply proportional variations of the contact siffness.
  - `Mu` to apply proportional variations of the friction coefficient.
  - `Omega` to apply proportional variations of the sliding velocity with `Euler` formulation.
- Only `struct` and `string` format definitions of parameter entries are accepted.

Zeto thickness elements are also supported. The same syntax can be used. Parameter types in field `.infp` are specific as

- `Kn` to apply proportional variations of the normal siffness.
- `Kt` to apply proportional variations of the planar siffness.
- `KnKt` to apply both `Kn` and `Kt` at once.

`TieSe`

This command generates a surface elastic bonding superelement, that corresponds to TIE or GLUE notions in most codes. It takes the same input as a `ctc_utils`Generate call but will either output a coupling superelement with command `TieSe` or the model to which the superelement has been added with command `TieSeAdd`.
Syntax is `SE=ctc_utils('TieSe',model,RA);`
First input is a finite element model, second input is a struture `RA` with mandatory fields `.slave` and `.msster` providing surface selections on which contact will be formulated. `RA` can contain any contact property field handled by `ctc_utils`Generate, so that the assembled superelement can integrate the desired formulation.

`Ctc2Zt`

This command replaces contact elements with zero thickness elements. `model=ctc_utils('Ctc2Zt',model`
The contact elements are replaced by zero thickness elements with `p_zt` properties. A `fe_casegConnection`
constraint is applied between the zero thickness elements and the slave surface. If the underlaying mesh is second order the new elements are also unjoined from the master surface and a `fe_casegConnectionSurface` constraint is applied between the zero thickness elements and the master surface. This is required as `p_zt` properties are applied to first order element topologies.
For bilinear contact properties the same contact stiffness density is applied, for other cases an automated estimation is performed. Tangential coupling properties are by default based on the

kappa values in p_contactlaws. No coupling is applied for models with no friction or no friction law.
The following command options are available

- -proid*val* to only convert elements with the specified ProId.
- -mtie to force unjoin and constraint the master surface side in all cases.

```
% Model with contact properties
model=d_contact('Cubes cbuild')
feutil('info',model)
% Convert to zt elements
model=ctc_utils('Ctc2Zt',model)
feutil('info',model)
```

MpcSe2Ctc

This command converts a coupling superelement based on a penalized fe_casegConnectionSurface constraint into a contact formulation. Contact elements are build on the master side of the constraint. By default contact properties are set as a bilinear contact with automatic stiffness density and a 2D sliding coulomb friction model with planar coupling for tangent models TangStick=1, contact states are evaluated on any present static state in the model stack curve,q0. It is possible to provide other properties.
The base syntax is model=ctc_utils('MpcSe2Ctc',model,list)
model is a model with superlements generated by penalized fe_casegConnectionSurface constraints (see feutilbCaseC2SE). list is a column cell array of superelement names to be converted. In option the second column of list can be a structure providing contact properties to be used as in GenerateContact command.
The following command options are available

- -useDOF to restrain original selections to elements that were effectively matched for the constraint.
- -flipSel to revert slave and master sides for contact in comparison to the constraint selections.
- -nocqinit not to setup the constact state estimation based on static states.

```
% Model to couple
model=d_contact('Cubes');
% Generate a penalized connection surface constraint
model=fe_caseg('ConnectionSurface -dens -KpAuto -MaxDist1e-6',model,'tie',...
 'group1 & selface & innode{z==1}','group2 & selface & innode{z==1}');
```

```
% compute modes
d1=fe_eig(model,[5 10 1e3]);
% convert penalized constraint to contact coupling
model=ctc_utils('MpcSe2Ctc',model,{'tie'});
% compute modes and check frequencies
d2=nl_solve('ModeSkip-fullDOF',model,[5 10 1e3]);
[d1.data d2.Mode.data(:,1)]
```

Zt2Ctc

This command converts zero thickness elements to contact elements. This is currently a partial implementation for normal contact without friction and `hexa8` based surface only. Please report to SDTools if a broader application case is necessary. A bilinear contact model based on `p_zt` properties is setup.
`model=ctc_utils('Zt2Ctc',model);`

TangentMdl

This command outputs a coupling superelement containing the tangent contact friction stiffnesses.
`SE=ctc_utils('TangentMdl',model,def);`
`model=ctc_utils('TangentMdl inModel',model);`
`model` is a standard SDT model, on which command `Generate` was performed, or an SDT/NL model with contact elements. `def` is a standard SDT deformation curve see `sdtweb def` which will be used to compute tangent coupling matrices. `def` should have a single column if a static state is provided. An optionally second column can be provided to specify a velocity.
The following command options allow specific use of the command

- `inModel` asks to output the input model with added coupling superelement relative to the tangent state, with name `tgtctc`.
- `-clean` combined with `inModel`, removes the contact elements and properties in the model, so that the output model only features an equivalent tangent coupling instead of contact.
- `-sepKj` asks to output the superelement with coupling matrices respectively split for contribution.
- `-setPar` for use in models with parametered contact and in conjunction with `inModel` to output a parametered tangent model.
- `-real` to generate a superelement with conservative matrix types only (adapted to real mode computations).
- `-evalFNL` to update non-linear static states based on stack entry `curve,q0` prior to generate the tangent model. This option is mandatory if the static state does not have a `.FNL` field.

Show

This commands performs visualization operations to display contact fields in the FEM. Contact fields are expressed at Gauss points, so that a triangulation of the contact Gauss points is necessary. This is performed by the GetObs command. If this was not performed earlier, Show will perform it itself. It can be noted that for SDT/NL models, this information may be precomputed (and stored in def.FNL.NL) by assigning parameter Sel to 1 in the contact parameter.
By default, the display units are MPa for pressures, and μm for lengths.
Command Show allows visualizing standard contact fields generated by SDT/contact module or SDT/NL, which are the following

- Pn Contact pressure, identified by the DOF .50.
- Ft1 Friction constraint in first local direction, identified by the DOF .51.
- Ft2 Friction constraint in second local direction, identified by the DOF .52.
- G Gap, identified by the DOF .54.
- W Sliding velocity, identified by the DOF .55.
- Pta Friction constraint norm, computed using Pt1 and Pt2.
- LMu Local Mu, computed as the ratio of Pta to Pn.

It must be noted that the model has to be contained in an feplot figure to allow display. If argument def is omitted, cf.def will be used.
The following command options can be used

- ProId*val* allows specifying the contact element of ProId *val* to be displayed. By default all groups are displayed.
- InMesh allows visualizing the contact fields in the full mesh in transparency. By default only the surfaces supporting contact data are displayed.
- unit*ID* allows specifying a unit different from the default one. The user must then specify the unit system *ID* (e.g. SI, or TM) in which the data has to be displayed.
- EltSel*EltSel_String*, in combination with InMesh allows specifying a mesh sub-selection in which the contact surfaces will be displayed, using a string *EltSel_String* that will be exploited by feutil('findElt*EltSel_String*,cf.mdl);.
- -reset Asks to reset the feplot display from scratch instead before generating the new display.
- Rel Asks to display the field variation relative to its initial state (first deformation column). This is useful to display small variations from a static state.

- Vel Asks to display the field velocity, computed by numerically deriving the provided field.

The following command lines illustrate how Show can be used

```
% display contact pressure
ctc_utils('showPn-reset',cf.mdl,def);
% display first direction firction constraint in the full mesh
ctc_utils('showFt1 InMesh',cf.mdl,def);
% display the friction constraint norm relative to its initial state
ctc_utils('ShowPta Rel -reset',cf.mdl,def);
% display the gap field in mm rather than in micrometers
ctc_utils('ShowG unit"MM"',cf.mdl,def);
```

Examples

Examples are provided in function d_contact.
A simple cube model and a simple disc brake models are available, respectively using the commands d_contact('cubes') and d_contact('DiscBrake').
Application examples are then available in the script command section:

```
% contact handling demonstration
d_contact('script Cube_Basic')
% disc brake with polar friction basis and convection observation
d_contact('script Brake Polar')
% simple static resolution
d_contact('script Brake')
```

p_contact,nl_contact

Purpose

Element property function for contact. This function is used by SDTools for internal developments and is not distributed with the base SDT. Integrated calls for high level handling are available using ctc_utils, this section aims at documenting the implementation details.

Syntax

```
 il=p_contact('dbval 1')
```

Description

For theory see section 3.1 , for implementation details, see section 3.2 . Contact is defined in SDT using contact elements. For 3D volumes, contact elements are then 3D shell elements. Since contact is non-linear, these elements are used in combination with the function nl_contact for computation of the contact force. See nl_spring, nl_solve to find out about non-linearities handling in the SDT/non-linear vibrations toolbox.

p_contact handles the contact formulation for integration, the il data is a structure that must be stored as a pro entry in the model Stack to comply with usual handling of non linear springs in SDT (through nl_spring).

Stack entry pro,Contact then follows the format

```
il = struct('type','p_contact',....
            'il', il,...
            'NLdata',NLdata,...
            'MAP',Map));
```

fields il, MAP, and NLdata are defined below.

```
il :  contact formulation parameters

    il =
    [ProID  fe_mat('p_contact','SI',1) ...
     Match Integ ContactLaw SlaveId FrictionLaw Euler TangStick LocalFric1stDir TOIM]
```

Physical parameters are stored in p_contact NLdata. The formulation options are as follows:

`ProID`	Element property identification number
`type`	Identifier obtained with `fe_mat('p_contact','SI',1)`
`Match`	Matching strategy (tolerance and choice of contact points). Format is *i.val*. *i* (integer part) select the matching strategy. 0 (default): Gauss integration point and matching to the volume underlying the slave surface, 1: Gauss point matching to the slave surface face mesh (quicker), -2 matches using the element nodes as contact point (**warning: relevant only when Integ is set at -2000 meaning nodal pressures**). *val* (digits after the comma) the tolerance token. Set to 0 uses the default matching radius of `1`. With `x.1` lets `p_contact` define the matching radius based on the slave surface mesh characteristic length (*e.g* set `Match` to `1.1` to use Gauss contact points matched to the surface with a radius based on the mesh). To customize the radius to `r` define val as *val*`= 10*r+9`. Thus `Match=0.209` to use Gauss contact points matched to the slave surface underlying volume with a radius of `20`. For radii less than 1 the same approach stands, as `1.029` with be used to define a radius of `0.2`.
`Integ`	Contact elements integration rule (see `integrules` for selection). It will determine the Gauss points used for contact. Adding `1000` to the `integrule` identifier allows activating a `CPress` operator converting contact resultants at node to contact pressures at nodes or contact points. This is currently used by `nl_contactl`. Adding `.1` to `Integ` will generate the observation based on the deformed mesh according to a deformation curved stored in the model stack as `curve,initContact`.
`ContactLaw`	contact law selection : `1` bilinear, `2` exponential pressure, `3` tabular, 4 power law, 9 external contact law function (user defined). Lagrange contact is defined in `nl_contactl`.

SlaveId	Identification of the set specifying the slave surface, by Id. The slave set data must then contain and ID field with a relevant value. For generation, see ctc_utils Generate or feutil AddSet.
FrictionLaw	Friction law selection : 0 none, 1 Coulomb unidirectional, 2 Regularized Coulomb unidirectional with a penalty slope, 3 Regularized Coulomb unidirectional with an arctangent function, 9 external friction law function (user defined). These are all 1D friction laws, to have their 2D equivalent, add 10 to the number: 11, 12, 13 and 19 are supported.
Euler	Euler formulation for the sliding velocity computation, to be used with friction. This adds a sliding velocity to the observation on the velocity field. This is useful to simulate displacements on fixed meshes. Use 1 for rotating bodies (this will add the Euler deformation of the meshes as well) or 2 for translating bodies. The Euler formulation assumes sliding is happening in the first friction direction, see MAP definitions below and p_shell.
TangStick	Sticking tangent state, for mode computation. If contact is stuck, you may want to add a symmetric stiffness coupling representing the impossibility for the system to slide in this direction for small vibrations. 0 none, 1 used.
LocalFric1stDir	Ordering of local friction directions. For each contact point, a local basis is defined, with the 3rd direction as the normal. This option allows customizing the ordering of the local friction direction. If set to 2 the friction directions are permuted and the new first friction direction orientation is made the same regardless of the normal direction. Is is possible to reverse the orientation by setting this value to -2. Set this to 0 to deactivate it.
TOIM	Topology Offset Induced Momentum, default to 0. If a clearance exists in the mesh topology between contact surface candidates, one can choose whether this has to be accounted for to define relative movement observation. If this is considered the observation will consider the clearance as a rigid section thus transmitting moments between the spaced contacting points. This feature is mostly useful when considering contact between shells, where one must account for the shell thickness. Set 0 to ignore (the default for solids), or 1 for use.

MAP : orientation for contact and friction

The contact normal is defined by the shell element normal. By default, friction directions are defined element-wisely depending on their orientation. If you want to control these directions, or even use customized local frames, a map needs to be defined, and added to the il data structure. These orientation maps are detailed in p_shell(of_mk('BuildNDN') integration rule 23).

For p_contact, the map third direction v3, (v3x,v3y,v3z) is fixed in the contact normal direction

(of the element). The first two directions `v1` and `v2` concern friction directions and can be modified. It typically suffices to define the first direction, the second one being completed to form a direct orthogonal frame. The first vector `v1x,v1y,v1z` is built in the direction of r lines (element edge defined) and `v2x,v2y,v2z` is tangent to the surface and orthogonal to `v1`.
For example, to fix the first sliding direction in the y direction of the global frame, and keep the third direction as the local normal, the map can be defined as

```
MAP=struct('dir',{{0,1,0,'v3x','v3y','v3z'}},...
           'lab',{{'v1x','v1y','v1z','v3x','v3y','v3z'}})
```

To define the first sliding direction at 45 degrees of the global frames x,z direction,

```
MAP=struct('dir',{{1/sqrt(2),1/sqrt(2),0,'v3x','v3y','v3z'}},...
           'lab',{{'v1x','v1y','v1z','v3x','v3y','v3z'}})
```

It is possible to define a field `Orig` in the format `[x0 y0 z0]` in the local contact MAP for use with the definition of rotation in the fixed frame `Euler=1`.

NLdata : Defining contact and friction laws parameters

The contact and friction laws (see section 3.3 for details) are handled by the function `nl_contact`. This is a specific non linear function, see `nllist`, and `nl_fun` for more details. The `NLdata` is a structure with fields

```
NLdata=struct('type','nl_contact',...
              'Fu','Kc 1e12',...
              'Fv','Mu 0 KtLin 50',...
              'cqoff',0,...
              'Vel', 0,...
              'KcLin', 1, ...
              'Sel' , 0 ,...
);
```

`type`	fixed value defining the non linearity type to be `nl_contact`
`Fu`	the contact law parameters, in a string format (defined below)
`Fv`	the friction law parameters, in a string format (defined below)
`cqoff`	offset for gap computations. This is a scalar added to the gap vector. To allow taking into account topology offsets (*e.g* mesh showing open contact), one can set `cqoff` either to `+Inf` to generate an offset vector based on the gap observation of the mesh with zero deformation, or to `-Inf` to generate an offset vector based on the positive only gap observation of the mesh with zero deformation.
`Vel`	sliding velocity. This is a scalar added to the sliding velocity vector when using Euler friction formulations.
`KcLin`	fixed Jacobian contact stiffness. This is a scalar defining a linear bilateral contact for the Jacobian computation in time simulations using the `NLNewmark` scheme of `fe_time`, to improve convergence.
`Sel`	to output the selection generating the gauss point mesh for contact field observation
`nodeVel`	an optional field to provide a restricted list of `NodeId` subjected to fixed frame motion (option `Euler`) if needed. The restriction is then applied to the contact points whose gap observation involves DOF relative to the nodes specified.

The contact law parameters `Fu` are defined in a string interpreted by `nl_contact` depending on the contact law specified in the `il` field.

The contact laws implemented deal with gap/pressure relationships to conform with the Signorini law.

The string is a succession of names and values representing the reserved parameter name and its associated value. The parameters interpreted depend on the contact law and are defined below. It is also possible to specify the internal laws parameters at low level by specifying the data vector specified.

1 linear	Parameter Kc giving the contact stiffness. *E.g.* `NLdata.Fu='Kc 1e12'` – or vector `[Kc]`.
2 exponential law	Parameter pz defines the contact pressure at zero gap and lambda defines the exponential coefficient. *E.g.* `NLdata.Fu='pz 1e2 lambda 75e4'` – or vector `[pz lambda]`.
3 tabular contact law	Parameter CurveId gives the curve ID (stacked in the model) giving the behavior. The curve is of the standard SDT format (sdtweb curve). With field X giving overclosures (opposite of the gap) and field Y giving the corresponding contact pressures. *E.g.* `NLdata.Fu='CurveId 1'` – or vector `[CurveId]`.
4: power contact law	Parameter pz defines the contact pressure at zero gap and parameter power defined the power used on the gap. The local contact pressure is then computed as $f_N = pz \cdot (-g)^{power}$ if $g < 0$ and 0 else (which will induce a positive contact force). The tangent state uses the derivative of the given power function. *E.g.* `NLdata.Fu='pz 1 power 3'` – or vector `[pz power]` .
9: external/user defined	The data string starts with the name of the non linear function called, followed by the parameters defined for the said nl_function, see nl_fun. *E.g.* `NLdata.Fu='nl_mycontact Kc 1e4'`. Note that the keywords Fu, Fc, cqoff, KcLin are reserved and must not be used in subfunctions. Parameter interpretation, may fail in such case.
9: Lagrange contact	This is an external non linear function,nl_contactl. This law does not take any parameter, such that the .Fu field need to be set as `NLdata.Fu='nl_contactl'`. Only a static model (based on the status method) has been implemented, to be solved with nl_solve Uzawa.

The bilinear law allows controlling the contact surface non-linear behavior for simplified studies. These variations are available with the string input format of the contact law definition. One can then use command option SymForceSet*val* to define a linearized behavior for transient and mode computation. The impacted elements can be restricted to a list defined by an EltId element set named SymForceSet stacked in the model.

- For SymForceSet*1*, the contact elements underlying the elements declared in set SymForceSet will be forced to contact and behave linearly, thus defining a partial symmetric bilinear coupling.

- For SymForceSet*2*, all contact elements follow a symmetric bilinear coupling.

- For SymForceSet*3*, the same transient behavior than for SymForceSet*1* is defined, but modes will be computed considering effective contact coupling exclusively with the declared elements in set SymForceSet.

- For SymForceSet*4*, all contact elements follow a symmetric bilinear coupling if any contact point is effectively in contact.

Eventually, using contact law 1000 will use the static state stacked in the model to generate a bilinear contact behavior for points into contact and no contact for opened points. This is mainly used to simplify the behavior in transient simulations, mostly for verification purposes.

The friction law parameters Fv are defined in a string interpreted by nl_contact depending on the friction law specified in the il field. The string is a succession of names and values representing the reserved parameter name and its associated value. The parameters interpreted depend on the friction law and are defined below. It is also possible to specify the internal laws parameters at low level by specifying the data vector specified.

0: no friction	the string can be empty (no interpretation) *E.g.* `NLdata.Fv=''`.
1, 11: Coulomb law	Parameter Mu gives the friction coefficient, optional parameter Nu gives the tangent sticking coefficient property, see section 3.3 .*E.g.* `NLdata.Fv='Mu 0.2'` – or vector `[Mu Nu]`.
2, 12: Regularized Coulomb , linear	Parameter Mu gives the friction coefficient, parameter KtLin gives the slope a zero sliding velocity, optional parameter Nu gives the tangent sticking coefficient property, see section 3.3 .*E.g.* `NLdata.Fv='Mu 0.2 KtLin 50 Nu 0.1'` – or vector `[Mu KtLin Nu]`.
3, 13: Regularized Coulomb , arctangent	Parameter Mu gives the friction coefficient, parameter KtLin gives the slope a zero sliding velocity, optional parameter Nu gives the tangent sticking coefficient property, section 3.3 . `NLdata.Fv='Mu 0.2 KtLin 50 Nu 0.1'` – or vector `[Mu KtLin Mu]`.
4, 14: Regularized Coulomb, scaled linear	Parameter Mu gives the friction coefficient, parameter KtLin gives the slope a zero sliding velocity, optional parameter Nu gives the tangent sticking coefficient property, see section 3.3 .*E.g.* `NLdata.Fv='Mu 0.2 KtLin 50 Nu 0.1'` – or vector `[Mu KtLin Nu]`.

9, 19 external/user defined	The data string starts with the name of the non linear function called, followed by the parameters defined for the said nl_function, see nl_fun. *E.g.* `NLdata.Fu='nl_myfriction Mu 0.4'`. The same reserved keywords than for external contact law definition apply.
9: Lagrange friction	This is an external non linear function, `nl_frictionl`. Parameter `Mu` gives the friction coefficient. The `.Fu` field needs to be set as `NLdata.Fu='nl_frictionl Mu 0.5'`. Only a static model (based on the status method) has been implemented, to be solved with `nl_solve Uzawa`.

Default values

It is possible to obtain default values for `NLdata` or `p_contact` data, using

```
% Standard il field, for ProId 1
il = p_contact('dbval 1 -struct')
% Standard il field with MAP defining first sliding direction along global x
il = p_contact('dbval 1 tan1dx -struct') % 1D friction
il = p_contact('dbval 1 tan2dx -struct') % 2D friction
% Standard il field with MAP defining first sliding direction along global y
il = p_contact('dbval 1 tan1dy -struct') % 1D friction
il = p_contact('dbval 1 tan2dy -struct') % 2D friction
% Standard il field with a polar MAP
il = p_contact('dbval 1 polar -struct')
% Standard il field with a polar MAP set at a custom origin [20 0 0]
il = p_contact('dbval 1 polar o 20 0 0 -struct')
```

Since the standardized default output of `dbval` commands is a `il` line vector, command option `-struct` has been added to allow `pro` entries under the struct format, to set in the model stack. Default MAPs are exploitable in the `dbval` command that can be used by specifying the MAP name in the `dbval` command. These MAPS are 2D maps in the friction plane,

- `tan1dx` or `tan2dx` defines the first friction direction to be along the x global axis, `1d` or `2d` specifies the number of friction directions considered.
- `tan1dy` or `tan2dy` defines the first friction direction to be along the y global axis, `1d` or `2d` specifies the number of friction directions considered.
- `polar` defines a polar MAP along the global frame origin.
- `polar o` *x0 y0 z0* defined a polar MAP with origin set to the point of coordinates [*x0 y0 z0*].

It is possible to obtain NLdata fields using nl_contact('db'). Without other information, a default field will be output, but it is also possible to give the wanted parameters. In this latter case, please note that all mandatory parameters must be specified.

```
% default field (ContactLaw=1, FrictionLaw=0);
NLdata=nl_contact('db');
% Parameters for a tabular contact law (3)
% and regularized Coulomb friction law (12)
NLdata=nl_contact(...
'db cqoff 0 Vel 0 KcLin 1e4 Fu''CurveId 18'' Fv''Mu 0.3 KtLin 50 Nu 0.25''');
```

The il, NLdata and MAP fields can be edited using ctc_utilsset command.

elementsDefining contact elements

A contact implementation requires the definition of a slave surface and a master surface. For the latter one, contact elements must be defined based on the underlying surface. Integrated calls can used with ctc_utilsGenerateContactPair (see example below). The steps necessary to define contact and the corresponding lower level calls are detailed in this section.
The slave surface must be stored in the model stack, with an explicit ID, as explained in the il definition. It can be generated using feutilb:

```
data=feutilb('FaceSet',model,EltFaceSel); % generate a FaceId set
data.ID=1;
model=stack_set(model,'set','slave',data); % stack in model
```

model is a standard SDT model and *EltFaceSel* is an element selector string (see sdtweb findelt), that must generate a surface selection. Command option -ID allows directly specifying a set ID.
The generation of the contact elements is handled by nl_contact, with call build:

```
model=nl_contact('build',model,EltFaceSel);
```

model is a standard SDT model and *EltFaceSel* is an element selector string (see sdtweb findelt), that must generate a surface selection. By default the contact elements ProId is incremented from the model. Note that no MatId is associated to contact elements.
Command option ProId*Val* allows specifying the contact elements ProId to *Val*.

Contact output

The non-linear contact forces are stored in the def.FNL field, see nl_spring. The non-linear contact DOF are defined using defined DOF extensions, .5, 51, .52, .53, .54, .55, respectively corresponding to the contact pressure F_N, the friction pressure in the first friction direction F_T1,

the friction pressure in the second friction direction `F_T2`, the Eulerian deformation `Utt` (geometric deformation in the first friction direction when `Euler` sliding is applied), the gap `G`, the sliding velocity norm `W`. Not all instances are always present depending on the chosen contact formulation. `.5, .53` and `.54` are at least always output.
This is a case where the same buffer is used for `.unl` and `.snl`. `.snl` contents are $F_N, F_{T1}, F_{T2}, U_{tt}$. The interlacing is also optimized for MATLAB operations and thus uses g at all Gauss points, the u_{T1} at all Gauss, ...

Time integration callbacks

- `AssembleCall` is used to generate proper assembly. The current nominal call is returned by `nl_spring('AssembleCall')`.

  The preparation is done in an Init and Exit phase. At the end of the Exit phase, `model.DOF=Case.DOF` and the original model DOFs are stored in `Case.mDOF` `assemble -fetime -matdes 2 3 1 -se load -initfcn "model=nl_spring(''InitTimeProp'',model);"`

- `OutputInit` see `sdtweb nl_spring('OutputInit')` handles preparation of output. `nl_spring('Out` is used for generation.

- `OutputInitCheck` see `sdtweb nl_spring('OutputInitCheck')` handles finalization of output data structure. This typically does low level verification needed for the optimization of integration.

- `OutputFcn` should be filled by `OutputInit`.

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

`Assembling strategy xxx`

build NL `sdtweb nl_spring('AssembleInit')`: scans all the non linear properties, disable the linear celas elements associated to non linear property, build the needed `matdes`.

`sdtweb nl_spring('AssembleExit')`: build the `model.NL` field (through a call to `nl_spring NL`).

Additional fields used for non-linear time integration

- `model.FNL` (low level vector used to pass data between functions)

  `model.FNLDOF`, `mo1.FNLlab` should be moved to output generation . Current FNLDOF is included in out.DOF.

  `sdtweb t_nlspring('of_time')` tests move to separate `out.FNL` which should become the reference.

  Generated by `nl_fun('init')` called by `nl_spring('nl -storefnl')`. xxx `-storefnl` should be done by non-linearity.
- `model.NL` stack of non linearities can be extended through `nl_fun('init')` calls.
-

Example

The following example creates a contact linearity between two cubes, computes a static solution and performs some contact field visualizations
First with an integrated call using `ctc_utils`

```
model=d_contact('Cubes -offset0');   % Load model
model=ctc_utils('GenerateContactPair',model,...
struct('ProId',201,'name','contactPro',...
'master','Group2 & SelFace & InNode{z==1}',...
'slave','Group1 & SelFace & InNode{z==1}',...
'contact','linear','Fu','Kc=1e11',...
'friction',0));
% compute a static solution
q0=nl_solve('static tol1e-9',model);
% perform some displays
cf=feplot(model,q0);
% show pressure field
ctc_utils('showPn-reset',cf.mdl,q0);
% show gap field in mm
ctc_utils('showG-unit"MM"inMesh',cf.mdl,q0);
% show pressure field in the mesh containing the top cube
ctc_utils('showPn inMeshEltSel"withnode{z>1.001}"',cf.mdl,q0)
```

Then with lower level calls

```
model=d_contact('cubes -offset0');   % Load model
% Generate slave surface (ID 301)
```

```
model=feutil('AddSetFaceId -ID301',model,'slave','Group1 & SelFace & InNode{z==1}');
% Generate contact elements (master), ProId 201
model=ctc_utils('build proid201',model,'Group2 & SelFace & InNode{z==1}');
% set pro, default
model=stack_set(model,'pro','contact',...
 p_contact('dbval 201 tan1dx slaveid 301 -struct'));
% compute a static solution
q0=nl_solve('static tol1e-9',model);
```

Example : Impact between two cubes

A usual example is the simulation of an impact between two solids. The behavior of such model is studied to gauge the integration scheme precision. The modified non linear Newmark scheme set here tackles down the problems commonly encountered with the basic Newmark scheme regarding the precision of the solution.

```
model=d_contact('cubes -cBuild');

% define the intial condition, with NL DOFs
[model,Case]=fe_case(nl_spring('AssembleCall'),model);
[i1,r1]=feutil('findnode group2',model);
q0=struct('DOF',Case.mDOF,'def',[zeros(size(Case.mDOF)) zeros(size(Case.mDOF))]);
q0.def(fe_c(Case.mDOF,[i1+.03],'ind'),2)=-.1;
model=stack_set(model,'curve','q0',q0);

% Modify the TimeOpt Newmark parameters
Opt=nl_solve('timeoptnlnewmark');
Opt.Opt=[.25 .5 0 2e-3 1e2];
Opt.RelTol=-1e-4;
model=stack_set(model,'info','TimeOpt',Opt);

% launch simulation and post treat contact data
def=fe_simul('time',model,model.Stack{end,3});
feplot(model,def); fecom colordataa
figure(4);
subplot(211);plot(def.data,fe_c(def.DOF,[70.03])*def.def);
xlabel('Time [s]'); ylabel('Node 65 disp.')
subplot(212);plot(def.data,fe_c(def.FNL.DOF,[.5])*def.FNL.def);
xlabel('Time [s]'); ylabel('fn at contact points [Pa]')
```

See also

nl_solve nllist nl_spring fe_timectc_utils, d_contact

Bibliography

rice_1983 [1] J. R. Rice and A. L. Ruina, “Stability of Steady Frictional Slipping,” *Journal of Applied Mechanics*, vol. 50, pp. 343–349, June 1983.

salencon83 [2] J. Salençon, *Viscoélasticité.* Presse des Ponts et Chaussés, Paris, 1983.

doghri00 [3] I. Doghri, *Mechanics of Deformable Solids.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2000.

simo00 [4] J. C. Simo and T. J. R. Hughes, *Computational Inelasticity.* No. 7 in Interdisciplinary Applied Mathematics Mechanics and Materials, New York, NY: Springer, 2 ed., 2000.

simo84 [5] J. C. Simo and R. L. Taylor, “Consistent tangent operators for rate-independent elastoplasticity,” *Computer methods in applied mechanics and engineering*, vol. 48, pp. 101–118, 1985.

schroder03 [6] J. Schröder and P. Neff, “Invariant formulation of hyperelastic transverse isotropy based on polyconvex free energy functions,” *International Journal of Solids and Structures*, vol. 40, pp. 401–445, Jan. 2003.

nria_2004 [7] INRIA and http://www.sdtools.com/pdf/sdt.pdfSDTools, *OpenFEM, a Finite Element Toolbox for Matlab and Scilab, http://www.openfem.netwww.openfem.net.* INRIA, Rocquencourt, SDTools, Paris, France, 2004.

R5.03.19 [8] M. Abbas, “Loi de comportement hyperélastique : matériau pres[...],” p. 8.

bal2 [9] *Structural Dynamics Toolbox (for Use with MATLAB).* Paris: SDTools, 1995/2020.

chapelle10 [10] D. Chapelle, J.-F. Gerbeau, J. Sainte-Marie, and I. E. Vignon-Clementel, “A poroelastic model valid in large strains with applications to perfusion in cardiac modeling,” *Computational Mechanics*, vol. 46, pp. 91–101, June 2010.

arckmann06 [11] G. Marckmann and E. Verron, “Comparison of hyperelastic models for rubber-like materials,” *Rubber Chemistry and Technology*, vol. 79, no. 5, pp. 835–858, 2006.

dal19 [12] H. Dal, Y. Badienia, K. Açikgöz, F. A. Denlï, Y. Badienia, K. Açikgöz, and F. A. Denlï, "A comparative study on hyperelastic constitutive models on rubber: State of the art after 2006," in *Constitutive Models for Rubber XI*, June 2019.

carroll11 [13] M. M. Carroll, "A Strain Energy Function for Vulcanized Rubbers," *Journal of Elasticity*, vol. 103, pp. 173–187, Apr. 2011.

ewicz_1989 [14] O. Zienkiewicz and R. Taylor, *The Finite Element Method.* MacGraw-Hill, 1989.

R3.06.08 [15] M. Abbas, "Finite elements treating the quasi-incompressibility," *Machine Translation*, p. 21.

avlev_2017 [16] R. Zhuravlev, *Contributions to the study of the mechanical behavior of railway tracks. Component with non-linear and dissipative behavior.* PhD thesis, ENSAM, Dec. 2017.

penas21 [17] R. Penas, E. Balmes, and A. Gaudin, "A unified non-linear system model view of hyperelasticity, viscoelasticity and hysteresis exhibited by rubber," *Mechanical Systems and Signal Processing*, vol. 170, p. 25, 2022.

nese_penas [18] R. Penas, *Models of Dissipative Bushings in Multibody Dynamics.* PhD thesis, Ecole Nationale Supérieure d'Arts et Métiers Paris, Nov. 2021.

ermot_2010 [19] 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, 2011.

umouille11 [20] V. Jaumouillé, *Dynamique des structures à interfaces non linéaires : Extension des techniques de balance harmonique.* PhD thesis, Ecole Centrale de Lyon, Mar. 2011.

nlvibkit14 [21] A. Sénéchal, B. Petitjean, and L. Zoghaib, "Development of a numerical tool for industrial structures with local nonlinearities," in *Proceedings of ISMA 2014 - International Conference on Noise and Vibration Engineering and USD 2014 - International Conference on Uncertainty in Structural Dynamics*, pp. 3111–3126, 2014.

nami_2014a [22] C. Hammami, *Intégration de Modèles de Jonctions Dissipatives Dans La Conception Vibratoire de Structures Amorties.* PhD thesis, Arts et Metiers ParisTech, Paris, Oct. 14.

gre02 [23] J. A. Greenwood and J. B. P. Williamson, "Contact of nominally flat surfaces," *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 295, no. 1442, pp. pp. 300–319, 1966.

moi1 [24] F. Moirot, *Etude de la stabilité d'un équilibre en présence de frottement de Coulomb.* PhD thesis, Ecole Polytechnique, 1998.

lor1 [25] X. Lorang, *Instabilité vibratoire des structures en contact frottant: Application au crissement des freins de TGV.* PhD thesis, Ecole Polytechnique, 2007.

`vol01` [26] D. Vola, M. Raous, and J. A. C. Martins, "Friction and instability of steady sliding: squeal of a rubber/glass contact," *Int. J. Numer. Meth. Engng.*, vol. 46, pp. 1699–1720, 1999.

`bau2` [27] T. Baumberger and C. Caroli, "Solid friction from stick-slip to pinning and aging," *Advances in Physics*, vol. 55, pp. 279–348, May 2006.

`biw01` [28] S. Biwa, S. Hiraiwa, and E. Matsumoto, "Stiffness evaluation of contacting surfaces by bulk and interface waves," *Ultrasonics*, vol. 47, no. 1 4, pp. 123 – 129, 2007.

`ver09` [29] 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.