Appendix C
NLvib: A Matlab Tool for Nonlinear Vibration Problems

Abstract NLvib is a free MATLAB tool for the computational analysis of nonlinear vibrations using Harmonic Balance, the shooting method and numerical path continuation. The intent of this appendix is to help getting started with NLvib. A certain level of experience with this tool is required to follow the solved exercises and tackle the homework problems in Chap. 5. We explain the rationale behind the code, describe how mechanical systems and nonlinearities are defined, and how the main functions and analysis types are used. An overview of the provided basic examples is also given.

Availability

NLvib is available via www.ila.uni-stuttgart.de/nlvib, including the MATLAB source code, examples and documentation. The copyright of NLvib rests with the authors of this book. The tool comes with absolutely no warranty. It is a free software, and can be redistributed and/or modified under the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. For details, see http://www.gnu.org/licenses or the file gpl-3.0.txt provided with the tool package.

Range of Capabilities

The tool is specifically designed for classroom use. Our goal was to find a reasonable compromise between simplicity and a broad portfolio of capabilities. At the same time, it is fairly easy to extend and specialize into various directions, some of which are indicated in the homework problems. In its initial version, the tool has the following capabilities:

- *Harmonic Balance* (HB): Alternating Frequency–Time HB.
- *numerical integration*: unconditionally stable Newmark integrator, shooting method.
- *numerical solution and path continuation*: predictor-corrector method with Newton-type solver (MATLAB's fsolve) and analytical gradients.

name	n	HB	Shooting	FRF	NMA
01_Duffing	1	o	-	o	-
02_twoDOFoscillator_cubicSpring	2	o	o	o	-
03_twoDOFoscillator_unilateralSpring	2	o	o	o	-
04_twoDOFoscillator_cubicSpring_NM	2	o	o	-	o
05_twoDOFoscillator_tanhDryFriction_NM	2	o	o	-	o
06_twoSprings_geometricNonlinearity	2	o	-	o	o
07_multiDOFoscillator_multipleNonlinearities	3	o	-	o	-
08_beam_tanhDryFriction	16	o	o	o	-
09_beam_cubicSpring_NM	38	o	-	-	o

Run times depend on your computing environment, but should not exceed a minute per example for a standard computer (2017).

n: number of degrees of freedom
HB: Harmonic Balance
FRF: nonlinear frequency response analysis
NMA: nonlinear modal analysis

Fig. C.1 Overview of basic examples included in the initial NLvib package

- *types of nonlinearities*: local generic nonlinear elements; distributed polynomial stiffness nonlinearity.
- *types of analysis*: nonlinear frequency response analysis; nonlinear modal analysis.

Several basic examples are provided with the tool. These are designed to cover most of NLvib's features, and are ordered with increasing degree of complexity. They are summarized in Fig. C.1.

Defining Mechanical Systems and Nonlinear Elements

The general equation of motion considered in NLvib is (c.f. Eq. 3.1)

$$\boldsymbol{M}\ddot{\boldsymbol{q}} + \boldsymbol{D}\dot{\boldsymbol{q}} + \boldsymbol{K}\boldsymbol{q} + \boldsymbol{f}_{\mathrm{nl}} = \Re\{\hat{\boldsymbol{f}}^{\dagger}_{\mathrm{ex}}(1)\mathrm{e}^{\mathrm{i}\Omega t}\}\,.$$

Herein, the †-representation is used, with $\hat{\boldsymbol{f}}^{\dagger}_{\mathrm{ex}}(1) = 2\hat{\boldsymbol{f}}_{\mathrm{ex}}(1) = \hat{\boldsymbol{f}}_{\mathrm{ex,c}}(1) - \mathrm{i}\hat{\boldsymbol{f}}_{\mathrm{ex,s}}(1)$, as introduced in Chap. 2 (c.f. Eq. 2.75). For simplicity, the code comes with only harmonic forcing. Note that you can easily generalize the external force to multiple harmonics. This is actually a good exercise to get familiar with the code. NLvib is clearly designed for mechanical systems whose equations of motion are usually written as second-order ordinary differential equations. As discussed in Chap. 3, however, additional first-order differential or algebraic equations can be considered with HB (in this case $\boldsymbol{M}$, or both $\boldsymbol{M}$ and $\boldsymbol{D}$, are singular, respectively).

Several classes are available for constructing and storing the coefficient matrices, define nonlinearities and the excitation. Object-oriented programming is not always very popular among new programmers. There is *absolutely no need* to use any of the *classes*, as shown in the following. We later explain how using specific classes can greatly simplify the treatment of certain mechanical systems.

Avoiding Classes

In this case, one directly provides the matrices $\boldsymbol{M}$, $\boldsymbol{D}$, $\boldsymbol{K}$, a description of the nonlinear elements and the excitation. One has to store this information in a structure array with the fields

- n (number of degrees of freedom),
- M,
- D,
- K, and
- nonlinear_elements.

M, D, K are real-valued n by n matrices. For a nonlinear frequency response analysis, the additional field Fex1 is needed, which is the vector $\hat{\boldsymbol{f}}_{\mathrm{ex}}^{\dagger}(1)$ in the equation of motion above. The structure array is provided to the solver, as explained later. The content of the field nonlinear_elements defines the term $\boldsymbol{f}_{\mathrm{nl}}$ and is described next.

Defining Nonlinear Elements

The field nonlinear_elements is a cell array of structure arrays. Each cell describes a nonlinear element. This makes it easy to consider many nonlinear elements in the same mechanical system.

Each nonlinear element's structure array must contain the fields

- type,
- islocal, and
- ishysteretic.

The field type is a character array that uniquely identifies the type of nonlinear element i.e., which specific force law is used to evaluate the contribution to the nonlinear force vector $\boldsymbol{f}_{\mathrm{nl}}$. The actual definition of the force laws is contained in the main functions HB_residual or shooting_residual. The fields islocal and ishysteretic are Boolean, and their meaning is described below.

Further fields are usually added for the specific parameters of each nonlinear element, e.g., stiffness for a nonlinear stiffness parameter, gap for a unilateral spring and so on. The field names for such parameters must be unique but are otherwise arbitrary.

Local Nonlinear Elements

Most currently implemented nonlinear elements are *local nonlinear elements*. These depend on a single input displacement (and velocity) and deliver a single output force to the system. The nonlinear force vector $\boldsymbol{f}_{\mathrm{nl}}$ of a system that has only local nonlinear elements is given by

$$\boldsymbol{f}_{\mathrm{nl}} = \sum_{e} \boldsymbol{w}_e \; f_{\mathrm{nl},e}\left(\boldsymbol{w}_e^{\mathrm{T}}\boldsymbol{q},\; \boldsymbol{w}_e^{\mathrm{T}}\dot{\boldsymbol{q}}\right) \, . \tag{C.1}$$

Herein, $f_{\mathrm{nl},e}(q_{\mathrm{nl}}, \dot{q}_{\mathrm{nl}})$ is a nonlinear force law. It must be scalar, but can otherwise be an arbitrary function $f_{\mathrm{nl},e} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. The input displacement and velocity are $q_{\mathrm{nl}} = \boldsymbol{w}_e^{\mathrm{T}} \boldsymbol{q}$ and $\dot{q}_{\mathrm{nl}} = \boldsymbol{w}_e^{\mathrm{T}} \dot{\boldsymbol{q}}$, respectively. $\boldsymbol{w}_e \in \mathbb{R}^{n\times 1}$ is the force direction of nonlinear element e. For local nonlinear elements, the additional field force_direction has to be provided to the respective cell within nonlinear_elements. We explain how $\boldsymbol{w}_e$ can be determined in the following.

In many cases, the input displacement (or velocity) is simply one component of the vector $\boldsymbol{q}$ (or $\dot{\boldsymbol{q}}$) and the output force acts on the specific coordinate. Then, $\boldsymbol{w}_e$ is simply a unit vector with a 1 at the corresponding coordinate index and 0 everywhere else. In other cases, the nonlinear element is applied between two coordinates, so that the input displacement is the difference between two components of the vector $\boldsymbol{q}$. Then, $\boldsymbol{w}_e$ contains one component with value $+1$ and one with value -1, and otherwise zeros. Suppose that a coordinate transformation of the form $\boldsymbol{q}_{\mathrm{old}} = \boldsymbol{P}\boldsymbol{q}_{\mathrm{new}}$ is applied with $\boldsymbol{P} \in \mathbb{R}^{n\times m}$, $n \geq m$. Then $\boldsymbol{w}_e^{\mathrm{T}}$ has to be replaced by $\boldsymbol{w}_e^{\mathrm{T}}\boldsymbol{P}$ in Eq. (C.1) (and accordingly $\boldsymbol{w}_e$ by $\boldsymbol{P}^{\mathrm{T}}\boldsymbol{w}_e$). This is relevant, for instance, when a modal transformation/truncation, or component mode synthesis is applied. Then, $\boldsymbol{w}_e$ is generally no longer a unit vector or a signed Boolean, but every element contains a certain real number.

NLvib uses the AFT scheme to determine the Fourier coefficients of the nonlinear forces. The AFT scheme is implemented in the function HB_nonlinear_forces_AFT contained in the file HB_residual.m. Depending on the type of nonlinear element, the specific force law $f_{\mathrm{nl},e}(q_{\mathrm{nl}}, \dot{q}_{\mathrm{nl}})$ is evaluated:

```
%% Evaluate nonlinear force in time domain
        switch lower(nonlinear_elements{nl}.type)
            case 'cubicspring'
                fnl = nonlinear_elements{nl}.stiffness*qnl.^3;
                dfnl = ...
            case 'mynewnonlinearity'
                fnl = ...
                dfnl = ...
            ...
```

For the shooting method, a similar distinction by the type of nonlinearity is implemented in the file shooting_residual.m. Herein, dfnl is an analytical expression for the first-order derivative of the nonlinear force. The analytical calculation of derivatives in general is discussed in Sect. 4.3. At the end of this appendix, we explain how these can be easily implemented. Several nonlinear force laws are already available in the public version of NLvib. These include a cubic spring, a unilateral spring, a regularized Coulomb dry friction law, and a viscous damper with quadratic dependence on the displacement (van-der-Pol-type). How to use them, and what specific parameters have to be provided, can be most easily learned by the basic examples. The user can add new types of nonlinear elements. To this end, it is good practice to copy the definition of the most closely related nonlinear element, specify a new type name and adjust the force law (including analytical gradients, if needed).

Tip
When you add a new nonlinearity, run the solver with jac option none. If everything is working properly, you can later accelerate the code by providing (correct) analytical derivatives. If you encounter convergence problems at that point, your derivatives are most likely wrong.

For many nonlinear elements, the force can be given as an explicit function of input displacement and velocity. However, some nonlinear elements describe a *hysteretic* process, where the current force value also depends on its time history. The treatment of hysteretic effects in general is explained in Sect. 3.3. An important example is the elastic dry friction element already implemented in NLvib. To remove the memory effect, one can introduce additional state variables and differential and/or algebraic equations to the dynamic force equilibrium. As explained in Chap. 3, an alternative is to apply the AFT scheme for more than one period until a steady force–displacement hysteresis cycle is reached. For the elastic dry friction element, two periods are usually sufficient, and NLvib currently uses two periods if the associated nonlinear element has ishysteretic set to 1 (or true). The basic example 07_multiDOFoscillator_multipleNonlinearities shows, among other aspects, how elastic dry friction elements are defined and used.

Global Nonlinear Elements

If a nonlinear force term cannot be brought into the form of Eq. (C.1), it is called global nonlinear element. An important example is a polynomial stiffness nonlinearity. Here, the i-th element of the nonlinear force vector is

$$f_{\mathrm{nl},i} = \sum_k E_{ki} \prod_j q_j^{p_{kj}} \,, \tag{C.2}$$

with the coefficients E_{ki} and exponents p_{kj}. The matrices $[E_{ki}]$ and $[p_{kj}]$ are real-valued n_z-by-n matrices, where n_z is the number of different polynomial terms. The polynomial stiffness nonlinearity can be useful for modeling mechanical systems with *kinematic nonlinearities*.

In the same model, both global and local nonlinear elements can be considered, of course. The extension to other nonlinearities, e.g. three-dimensional contact elements, is known to be possible from existing implementations not included in the initial tool package. Such extensions are deliberately left to the prospective users, who hopefully regard this as a suitable task for making themselves familiar with NLvib.

Mechanical System Class

Although this is not mandatory, we recommend using the available classes for representing mechanical systems. Using classes has certain advantages: First, a class can be defined in such a way that it takes care of setting reasonable default values for certain parameters. Note that some parameters are only relevant in special cases or have trivial values in many situations. For example, the default value for the ishysteretic flag is false. Single-DOF oscillators are internally handled as special case of multi-DOF oscillators, so that a force direction has to be given for each nonlinear element. The single-DOF oscillator class automatically sets this to 1 (which is the only reasonable value in this case). When default values are used, they do not have to be explicitly provided, which can significantly simplify the definition of mechanical systems by the user. Another important advantage of using classes is that during the construction of an object, the consistency of provided data can be checked. For example, it could be checked if the provided matrices $\boldsymbol{M}$, $\boldsymbol{D}$, $\boldsymbol{K}$ have the expected dimensions and are real-valued.

Using classes is not much more complicated than using structures: Most of the time, the class just stores the properties, just as a structure array does. In fact, the syntax to access properties of an object (a particular realization of a class) is exactly the same as to access an array of a structure array in Matlab.

The most generic class for mechanical systems in NLvib is the MechanicalSystem class implemented in the file MechanicalSystem.m. This code shows how to define an object of this class:

```
% Define properties
M = ...      % n x n matrix
D = ...       % n x n matrix
K = ...       % n x n matrix
Fex1 = ...% n x 1 vector
w1 = ...    % n x 1 vector
...

% Define nonlinear elements
nonlinear_elements{1} = struct('type','mytype',...
        'force_direction',w1,['p1',v1,'p2',v2,...]);
nonlinear_elements{2} = ...

% Define mechanical system
mySystem = MechanicalSystem(M,D,K,...
        nonlinear_elements,Fex1);
```

Herein, p1, v1, p2, v2, ...are parameter-name-and-value combinations. These are optional arguments in general, which is denoted by the brackets [ ] here and in the following. For a specific nonlinearity it might, of course, be mandatory to provide certain parameters.

A few other classes are available: a single mass oscillator, a chain of oscillators, finite element models of beams and rods, a systems with polynomial stiffness nonlinearity. These are subclasses of the parent class MechanicalSystem, which means that they inherit the way one interacts with them as user (via properties and methods). These derived classes are described in the following.

Single Mass Oscillator Class

This class is defined in the file SingleMassOscillator.m. It is used in some of the solved exercises and homework problems in Chap. 5. It is relatively straightforward to use as shown in this code:

```
% Define properties
m = ...
d = ...
k = ...
Fex1 = ...
...

% Define nonlinear elements
nonlinear_elements{1} = ...

% Define single mass oscillator
myOscillator = SingleMassOscillator(m,d,k,...
        nonlinear_elements,Fex1);
```

The single mass oscillator is a special case of a chain of oscillators, which is described next.

Chain of Oscillators Class

This class is defined in the file ChainOfOscillators.m. The class is used from the second through the fifth basic example (c.f. Fig. C.1). It is a chain of masses having one degree of freedom each as illustrated in Fig. C.2. Each mass is connected via springs to its nearest neighbors, the first and the last mass are connected to the ground. This code shows how to define a chain of oscillators:

```
% Define properties
mi = ...      % row or column vector with length n
ki = ...       % row or column vector with length n+1
di = ...       % row or column vector with length n+1
Fex1 = ... % n x 1 vector
...

% Define nonlinear elements
nonlinear_elements{1} = ...

% Define chain of oscillators
myChain = ChainOfOscillators(mi,di,ki,...
        nonlinear_elements,Fex1);
```

Using the coordinates $q_1, \dots, q_n$ indicated in Fig. C.2, the mass and stiffness matrices are

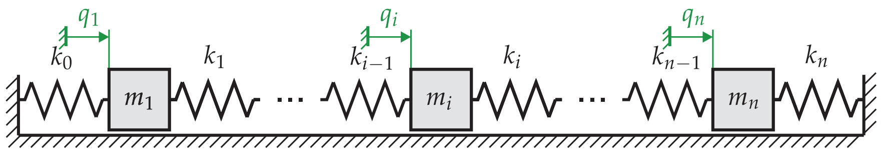

Fig. C.2 Illustration of a chain of oscillators

$$
\boldsymbol{M} = \mathrm{diag}[m_i]\,,
$$

$$
\boldsymbol{K} = \begin{bmatrix} k_0+k_1 & -k_1 & 0 & \cdots & 0 \\ -k_1 & k_1+k_2 & -k_2 & \ddots & \vdots \\ 0 & -k_2 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -k_{n-1} \\ 0 & \cdots & 0 & -k_{n-1} & k_{n-1}+k_n \end{bmatrix}.
$$

In parallel to the springs, viscous dampers can be introduced. The resulting matrix $\boldsymbol{D}$ has a form analogous to $\boldsymbol{K}$. If there is no spring between the first mass and the ground, $k_0 = 0$. Similarly, $k_n = 0$, if the last mass is not connected to the ground, and $k_i = 0$ with $0 < i < n$ if there is no spring between mass m_i and mass m_{i+1}.

The intent of the class is to help setting up the coefficient matrices for chains of oscillators. It is important to note that once the ChainOfOscillators object is constructed, the matrices $\boldsymbol{M}$, $\boldsymbol{D}$, and $\boldsymbol{K}$ are properties of the object and can be modified. For instance, one can easily add further springs between an intermediate mass and the ground, or springs between masses that are not adjacent to each other.

The coordinates can be understood as generalized coordinates. Hence, the chain of oscillators class can be used to model not only chains with translational displacement degrees of freedom, but also chains of rotating inertias with rotational springs.

FE Model of a Euler–Bernoulli Beam

A simple finite element model of a Euler–Bernoulli beam, as depicted in Fig. C.3, is available in NLvib. Each node i has two degrees of freedom, one for the bending displacement w_i and one for the slope w_i'. The class is defined in FE_EulerBernoulliBeam.m. This code shows how to define such a system:

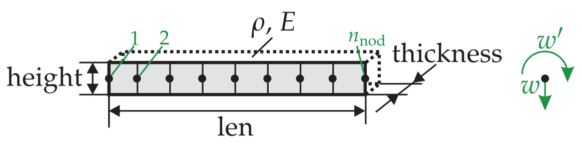

Fig. C.3 Illustration of the FE model of a Euler–Bernoulli beam

```
% Define properties
...
BCs = 'clamped-free';  % example with clamping on
    the left and free end on the right; pinned is
    also possible; arbitrary combinations are
   allowed
n_nod = ...            % number of nodes

% Define beam (rectangular cross section)
myBeam = FE_EulerBernoulliBeam(len,height,...
        thickness,E,rho,BCs,n_nod);

% Apply external forcing (works in an additive
   way)
inode = ...                  % node index
dof = ...                    % degree of freedom
   specifier ('rot' or 'trans')
Fex1 = ...                   % complex-valued scalar
add_forcing(myBeam,inode,dof,Fex1);

% Apply nonlinear attachment (only grounded
   elements)
inode_nl = ...               % see above
dof_nl = ...                 % see above
add_nonlinear_attachment(myBeam,inode,dof,...
        type,['p1',v1,'p2',v2,'...']);
```

n_nod is the number of nodes, n_{nod}. The boundary conditions are specified by a character array. free, pinned, and clamped conditions are implemented. These can be set independently on the left and on the right end to allow arbitrary combinations. The other parameters are (in this order): length, height, thickness (assuming a rectangular cross section), Young's modulus, and density.[1]

The beam class has a method for applying external forces and moments concentrated at any node. This method works in an additive way; i.e., if one applies multiple loads at the same node in the same direction, the loads are added (as opposed to overwritten).

The beam class has a method for adding nonlinear elements between any degree of freedom of any node and the ground. As in the case of the external loading, this method works in an additive way. If a nonlinear element shall be introduced among different nodal degrees of freedom, then the associated nonlinear element has to be defined manually, in the lower level way described earlier.

[1]Note that in accordance with Euler–Bernoulli theory, only the products ρA (mass per unit length) and EI (bending stiffness) are relevant. This can be useful for defining beams with non-rectangular, e.g., circular cross section. The reason why these parameters are to be specified individually is to make it easier to extend the class by capabilities that require the actual geometry and material properties (e.g. two-dimensional animation, stress analysis).

A property of the class is the matrix $\boldsymbol{L}$, which is automatically generated when the beam object is constructed. This matrix is useful to recover *all nodal degrees of freedom* (including those constrained by boundary conditions) once $\boldsymbol{q}$ is known,

$$\boldsymbol{q}_{\text{full}} = \begin{bmatrix} w_1 \\ w_1' \\ \vdots \\ w_{n_{\text{nod}}} \\ w_{n_{\text{nod}}}' \end{bmatrix} = \boldsymbol{L}\boldsymbol{q} \, .$$

$\boldsymbol{L}$ contains columns of the identity matrix. Those columns which are associated to a constrained degree of freedom are automatically removed and not retained in $\boldsymbol{L}$. This ensures that any motion of the generalized coordinates $\boldsymbol{q}$ is compatible with the boundary conditions.

FE Model of an Elastic Rod

The rod class is very similar to that of the beam. The one-dimensional finite element model of the rod is illustrated in Fig. C.4. Note that each node i of a rod has only a translational degree of freedom x_i. The class is defined in FE_ElasticRod.m. This code shows how to define such a system:

```
% Define properties
...
BCs = 'pinned-free';    % example with pinning on the
     left and free end on the right; arbitrary
    combinations are allowed
n_nod = ...             % number of nodes

% Define rod
myRod = FE_ElasticRod(len,A,E,rho,BCs,n_nod);

% Apply external forcing (works in an additive way)
inode = ...                  % node index
Fex1 = ...                   % complex-valued scalar
add_forcing(myRod,inode,Fex1);

% Apply nonlinear attachment (only grounded elements
    )
inode_nl = ...               % see above
add_nonlinear_attachment(myRod,inode,...
        type,['p1',v1,'p2',v2,'...']);
```

Analogous to the beam, the matrix $\boldsymbol{L}$ takes care of the transform

$$\boldsymbol{q}_{\text{full}} = \begin{bmatrix} x_1 \\ \vdots \\ x_{n_{\text{nod}}} \end{bmatrix} = \boldsymbol{L}\boldsymbol{q} \, .$$

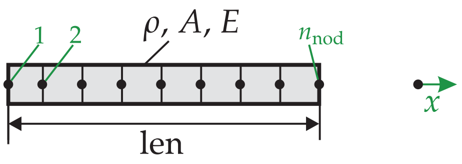

Fig. C.4 Illustration of the FE model of an elastic rod

System with Polynomial Stiffness

This class is designed to model systems with polynomial stiffness nonlinearity as defined in Eq. (C.2). The class is defined in System_with_PolynomialStiffness Nonlinearity.m. This code shows how to define such a system:

```
% Define properties
p = ... % Nz x n vector of nonnegative integers
E = ... % Nz x n vector of real-valued coefficients
...

% Define system
myPolyStiffSys =
    System_with_PolynomialStiffnessNonlinearity...
         (M,D,K,p,E,Fex1);
```

To elucidate the meaning of the coefficients and exponents, consider the system with geometric nonlinearity shown in Fig. C.5. This system is defined in the basic example 06_twoSprings_geometricNonlinearity. Its equation of motion can be approximated as

$$\begin{aligned}
&\ddot{q}_1 + 2\zeta_1\omega_1\dot{q}_1 + \omega_1^2 q_1 \\
&\quad + \frac{3\omega_1^2}{2}q_1^2 + \omega_2^2 q_1 q_2 + \frac{\omega_1^2}{2}q_2^2 + \frac{\omega_1^2+\omega_2^2}{2}q_1^3 + \frac{\omega_1^2+\omega_2^2}{2}q_1 q_2^2 = 0\,, \\
&\ddot{q}_2 + 2\zeta_2\omega_2\dot{q}_2 + \omega_2^2 q_2 \\
&\quad + \frac{\omega_1^2}{2}q_1^2 + \omega_2^2 q_1 q_2 + \frac{3\omega_1^2}{2}q_2^2 + \frac{\omega_1^2+\omega_2^2}{2}q_1^2 q_2 + \frac{\omega_1^2+\omega_2^2}{2}q_2^3 = 0.
\end{aligned}$$

In this case, it makes sense to define $n_z = 7$ terms z_k, namely the three quadratic terms q_1^2, q_1q_2, q_2^2 and the four cubic terms q_1^3, $q_1^2q_2$, $q_1q_2^2$, q_2^3. From the z_k, one can

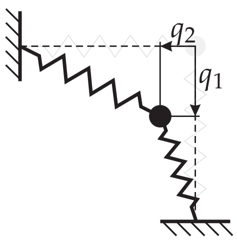

Fig. C.5 Example of a system with geometrical nonlinearity

easily identify the exponents p_{kj}. Suppose we define $z_1 = q_1^2 = q_1^2 q_2^0$. We then have $p_{11} = 2$, $p_{12} = 0$, and $E_{11} = 3\omega_1^2/2$, $E_{12} = \omega_1^2/2$.

Solve and Continue

The function in the file solve_and_continue.m takes care of the continuation problem

$$\begin{aligned} \text{solve} \quad & \boldsymbol{R}(\boldsymbol{X}) = \boldsymbol{0} \\ \text{with respect to} \quad & \boldsymbol{X} = \begin{bmatrix} \boldsymbol{x} \\ \lambda \end{bmatrix} \\ \text{in the interval} \quad & \min\left(\lambda^{\mathrm{s}}, \lambda^{\mathrm{e}}\right) \le \lambda \le \max\left(\lambda^{\mathrm{s}}, \lambda^{\mathrm{e}}\right) \ . \end{aligned}$$

The function is deliberately developed to be rather independent of the actual analysis to be carried out. It merely receives a handle to the function $\boldsymbol{R}(\boldsymbol{X})$, the range of the free parameter, λ, and an initial guess, $\boldsymbol{x}^{(0)}$, and then treats the problem as a black box. This permits reusing the same continuation code for a wide range of problems. Likewise, one does not have to take care of the parametrization of the solution branch (e.g. constraint equation $p_{\mathrm{c}}(\boldsymbol{X})$) within the implementation of the HB or shooting method.

This code shows the syntax:

```
[X,Solinfo,Sol] = solve_and_continue(x0,...
        fun_residual,lam_s,lam_e,ds,...
        [Sopt,fun_postprocess,opt_fsolve]);
```

Output arguments:

- X: columns are solution points.
- Solinfo: structure array with the fields FCtotal (total function evaluation count), ctime (total computation time of the continuation, measured with tic, toc), and per solution point it contains the data iEx (fsolve exit flag), NIT (number of iterations), FC (function evaluation count).
- Sol: array of structures returned by the optional postprocessing functions.

Input arguments:

- x0: initial guess.
- fun_residual: residual function R(X).
- lam_s: start value of free parameter λ^{s}.
- lam_e: end value of free parameter λ^{e}.
- ds: nominal step length of numerical path continuation.
- Sopt: continuation options structure (optional).
- fun_postprocess: cell array of postprocessing functions F(X), stored in output Sol (optional).
- opt_fsolve: fsolve options (optional).

The continuation always starts at λ^{s} and ends at λ^{e}. To continue backwards, one simply sets λ^{s} to the upper and λ^{e} to the lower interval endpoint of the free parameter. The initial guess has to be good for $\lambda = \lambda^{\mathrm{s}}$. If you want to specify fun_postprocess but not Sopt, simply place empty brackets ([ ]) instead of Sopt and so on.

The most common continuation options are (fields of the structure array Sopt):

- flag: Boolean 0 for only sequential continuation, 1 for actual path continuation (default).
- predictor: tangent or secant predictors can be specified (tangent (default) or secant).
- parametrization: definition of parametrization constraint $p_{\mathrm{c}}(\boldsymbol{X}) = 0$ (arc_length (default), pseudo_arc_length, local, or orthogonal).
- dsmin: minimum step length (default: ds/5).
- dsmax: maximum step length (default: ds*5).
- stepadapt: Boolean whether step length should be automatically adjusted (default: 1; recommended if flag = 1).
- stepmax: maximum number of steps before termination.
- termination_criterion: cell array of functions (X) returning logic scalar 1 for termination.
- jac: specifier whether the extended Jacobian $\partial\boldsymbol{R}/\partial\boldsymbol{X}$ is analytically determined (default: jac = full), only the Jacobian $\partial\boldsymbol{R}/\partial\boldsymbol{x}$ is analytically determined (jac = x), or a finite-difference approximation should be computed (jac = none).
- Dscale: vector of typical values/linear scaling values for vector $\boldsymbol{X}$, used for diagonal preconditioner, c.f. Sect. 4.3.

Analysis Types

Two analysis types are available:

1. Nonlinear frequency response analysis,
2. Nonlinear modal analysis.

For the former, a harmonic external forcing is assumed, $\boldsymbol{f}_{\mathrm{ex}}(t) = \Re\{\hat{\boldsymbol{f}}^{\dagger}_{\mathrm{ex}}(1)\mathrm{e}^{\mathrm{i}\Omega t}\}$. For the latter, the periodic motion definition of nonlinear modes is used, in its extended form for dissipative systems [2]. Both analysis types work with HB and shooting. This yields 4 different problem formulations (residual function $\boldsymbol{R}$, vector of unknowns $\boldsymbol{x}$, continuation parameter λ).

The HB residual function is implemented in the file HB_residual.m. This code shows the syntax:

```
[R,dR,Q] = HB_residual(X,system,H,N,...
    [analysis_type,varargin])
```

Output arguments:

- R: residual vector $\boldsymbol{R}$.
- dR: extended Jacobian $\partial\boldsymbol{R}/\partial\boldsymbol{X}$.

- Q: Fourier coefficients of $\boldsymbol{q}_{\mathrm{h}}$ (representation defined later).

Input arguments:

- X: vector of unknowns extended by continuation parameter, $\boldsymbol{X} = \left[\boldsymbol{x}^{\mathrm{T}}\ \lambda\right]^{\mathrm{T}}$
- system: description of the system, in the form of either a structure array or an object of class MechanicalSystem (or one of its subclasses).
- H: harmonic truncation order.
- N: number of time samples per period for the AFT scheme.
- analysis_type: character array that uniquely identifies the type of analysis, FRF for nonlinear frequency response, NMA for nonlinear modal analysis (optional input, default: FRF).
- varargin: additional input required for nonlinear modal analysis, see below.

The shooting residual function is implemented in the file shooting_residual.m. This code shows the syntax:

```
[R,dR,ye,Y,dye_dys] = ...
    shooting_residual(X,system,Ntd,Np,...
    analysis,[qscl,fscl,inorm])
```

Output arguments:

- R: residual vector $\boldsymbol{R}$.
- dR: extended Jacobian $\partial \boldsymbol{R}/\partial \boldsymbol{X}$.
- ye: vector of state variables at the end of the simulated time interval, $\boldsymbol{y}(t^{\mathrm{e}})$.
- Y: $N_{\mathrm{p}}N_{\mathrm{td}} \times 2d$ matrix whose rows are vectors of state variables for each time level (it is only stored if this output argument is requested).
- dye_dys: monodromy matrix $\boldsymbol{\Psi}(T)$.

Input arguments:

- X: vector of unknowns extended by continuation parameter, $\boldsymbol{X} = \left[\boldsymbol{x}^{\mathrm{T}}\ \lambda\right]^{\mathrm{T}}$.
- system: description of the system, in the form of either a structure array or an object of class MechanicalSystem (or one of its subclasses).
- Ntd: number of time levels per period.
- Np: number of periods for the shooting problem ($N_{\mathrm{p}} > 1$ could be useful to analyze subharmonic responses, or to use the function as usual integrator for a longer duration).
- analysis: character array that uniquely identifies the type of analysis, FRF for nonlinear frequency response, NMA for nonlinear modal analysis.
- qscl: optional input for improved preconditioning, see below.
- fscl: optional input for improved preconditioning, see below.
- inorm: additional input required for nonlinear modal analysis, see below.

Nonlinear Frequency Response Analysis (FRF)

For the nonlinear frequency response analysis, the continuation parameter is the excitation frequency, $\lambda = \Omega$.

Harmonic Balance

For HB, we have the following residual $\boldsymbol{R}$ and vector of unknowns $\boldsymbol{x}$:

$$\boldsymbol{R} = \begin{bmatrix} \hat{\boldsymbol{r}}(0) \\ \hat{\boldsymbol{r}}_{\mathrm{c}}(1) \\ \hat{\boldsymbol{r}}_{\mathrm{s}}(1) \\ \vdots \\ \hat{\boldsymbol{r}}_{\mathrm{c}}(H) \\ \hat{\boldsymbol{r}}_{\mathrm{s}}(H) \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} \hat{\boldsymbol{q}}(0) \\ \hat{\boldsymbol{q}}_{\mathrm{c}}(1) \\ \hat{\boldsymbol{q}}_{\mathrm{s}}(1) \\ \vdots \\ \hat{\boldsymbol{q}}_{\mathrm{c}}(H) \\ \hat{\boldsymbol{q}}_{\mathrm{s}}(H) \end{bmatrix}.$$

NLvib uses the sine–cosine representation externally (for $\boldsymbol{x}$ and $\boldsymbol{R}$). Internally, the †-variant of the complex-exponential representation is used (as introduced in Chap. 2, c.f. Eq. 2.73). To this end, the conversion rules in Eq. (2.75) are applied,

$$\hat{\boldsymbol{q}}^{\dagger}(0) = \hat{\boldsymbol{q}}(0)\,, \tag{C.3}$$

$$\hat{\boldsymbol{q}}^{\dagger}(k) = \hat{\boldsymbol{q}}_{\mathrm{c}}(k) - \mathrm{i}\hat{\boldsymbol{q}}_{\mathrm{s}}(k)\,, \quad k = 1, \dots, H\,. \tag{C.4}$$

After the Fourier coefficients of the nonlinear forces are computed with the AFT scheme, the residual is set up as

$$\begin{aligned} \hat{\boldsymbol{r}}^{\dagger}(k) = &\left(-(k\Omega)^2\boldsymbol{M} + \mathrm{i}k\Omega\boldsymbol{D} + \boldsymbol{K}\right)\hat{\boldsymbol{q}}^{\dagger}(k) \\ &+ \hat{\boldsymbol{f}}^{\dagger}_{\mathrm{nl}}(k) - \hat{\boldsymbol{f}}^{\dagger}_{\mathrm{ex}}(k)\,, \quad k = 0, \dots, H\,. \end{aligned} \tag{C.5}$$

Finally, the Fourier coefficients of the residual in the †-representation are converted back to the sine–cosine representation,

$$\hat{\boldsymbol{r}}(0) = \hat{\boldsymbol{r}}^{\dagger}(0)\,, \tag{C.6}$$

$$\hat{\boldsymbol{q}}_{\mathrm{c}}(k) = \Re\{\hat{\boldsymbol{r}}^{\dagger}(k)\}\,, \quad \hat{\boldsymbol{q}}_{\mathrm{s}}(k) = -\Im\{\hat{\boldsymbol{r}}^{\dagger}(k)\}\,, \quad k = 1, \dots, H\,. \tag{C.7}$$

Shooting Method

For the shooting method, we have the following residual $\boldsymbol{R}$ and vector of unknowns $\boldsymbol{x}$:

$$\boldsymbol{R} = \begin{bmatrix} (\boldsymbol{q}(T) - \boldsymbol{q}(0))\,\frac{1}{q_{\mathrm{scl}}} \\ \\ (\boldsymbol{u}(T) - \boldsymbol{u}(0))\,\frac{1}{\Omega q_{\mathrm{scl}}} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} \boldsymbol{q}(0) \\ \\ \frac{\boldsymbol{u}(0)}{\Omega} \end{bmatrix}.$$

After division by the oscillation frequency Ω, the rescaled velocities have the same physical unit and typical values as the generalized coordinates. Recall that $T = 2\pi/\Omega$ and $\boldsymbol{u} = \dot{\boldsymbol{q}}$. The positive real number q_{scl} should be a typical value for the generalized coordinates. In the rescaled formulation, the different elements of the residual, $\boldsymbol{R}$, should have similar order of magnitude, independent of the vibration level and the oscillation frequency. Again, the primary goal of this scaling is to improve the conditioning of the problem. Moreover, for small vibration levels, the residual is typically small, too. To distinguish typically small residuals from the sought solutions, one has to adjust the solver's tolerance. With the above-described scaling, the residual should always have the same typical values, so that the solver's tolerances can be kept constant. In NLvib, $\boldsymbol{q}(T)$, $\boldsymbol{u}(T)$ are determined by numerical forward integration from the initial values $\boldsymbol{q}(0)$, $\boldsymbol{u}(0)$ using the Newmark integrator described in the next section.

Nonlinear Modal Analysis (NMA)

For the nonlinear modal analysis, the continuation parameter is the logarithm of an amplitude quantity, $\lambda = \log_{10} a$.

Harmonic Balance

For HB, we have the following residual $\boldsymbol{R}$ and vector of unknowns $\boldsymbol{x}$:

$$\boldsymbol{R} = \begin{bmatrix} \hat{\boldsymbol{r}}(0)\,\frac{1}{f_{\mathrm{scl}}a} \\ \hat{\boldsymbol{r}}_{\mathrm{c}}(1)\,\frac{1}{f_{\mathrm{scl}}a} \\ \hat{\boldsymbol{r}}_{\mathrm{s}}(1)\,\frac{1}{f_{\mathrm{scl}}a} \\ \vdots \\ \hat{\boldsymbol{r}}_{\mathrm{c}}(H)\,\frac{1}{f_{\mathrm{scl}}a} \\ \hat{\boldsymbol{r}}_{\mathrm{s}}(H)\,\frac{1}{f_{\mathrm{scl}}a} \\ \Re\{\sum\limits_{k=0}^{H} \overline{\hat{\boldsymbol{q}}^{\dagger}(k)}^{\mathrm{T}} \boldsymbol{M}\hat{\boldsymbol{q}}^{\dagger}(k)\}/a^2 - 1 \\ \dot{q}_{i_{\mathrm{norm}}}(0)/(\omega a) \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} \hat{\boldsymbol{q}}(0)\,\frac{1}{a} \\ \hat{\boldsymbol{q}}_{\mathrm{c}}(1)\,\frac{1}{a} \\ \hat{\boldsymbol{q}}_{\mathrm{s}}(1)\,\frac{1}{a} \\ \vdots \\ \hat{\boldsymbol{q}}_{\mathrm{c}}(H)\,\frac{1}{a} \\ \hat{\boldsymbol{q}}_{\mathrm{s}}(H)\,\frac{1}{a} \\ \omega \\ \zeta \end{bmatrix}. \tag{C.8}$$

Herein, f_{scl} is a positive real-valued scalar, ω is the nonlinear modal frequency, ζ is the nonlinear modal damping ratio. The second-last equation takes care of the amplitude normalization. It defines a^2 as modal mass of $\hat{\boldsymbol{q}}^{\dagger}$ (all harmonics together). It is not sufficient to consider just the modal mass of the fundamental harmonic, since in the case of an internal resonance, this typically goes to zero, such that $\log_{10} a \to -\infty$. The last equation takes care of the phase normalization of the autonomous system. To this end, the index i_{norm} must be selected in such a way that $\dot{q}_{i_{\mathrm{norm}}}(t)$ has regular zero crossings, i.e., it does remain zero for a finite duration. For the nonlinear modal analysis, the HB residual is defined as in Eq. (C.5), only with $\hat{\boldsymbol{f}}^{\dagger}_{\mathrm{ex}}(k) = \boldsymbol{0}$ for all k, and with $\boldsymbol{D}$ replaced by $\boldsymbol{D} - 2\zeta\omega\boldsymbol{M}$. Again, the rationale behind scaling of the residual is to achieve similar orders of magnitude among the different components. Otherwise the dynamic force equilibrium or the normalization conditions would have unreasonably strong weight, which could have a detrimental effect on the convergence of the solver. To this end, f_{scl} should be a typical value of the forces divided by a. In our experience, the mean value of the vector $\left|\boldsymbol{K}\boldsymbol{\varphi}_j\right|$ works well, where $\boldsymbol{\varphi}_j$ is the mass-normalized eigenvector of the considered mode j in the linear case.

Shooting Method

For the shooting method, we have the following residual $\boldsymbol{R}$ and vector of unknowns $\boldsymbol{x}$:

$$\boldsymbol{R} = \begin{bmatrix} \left(\boldsymbol{q}(T) - \boldsymbol{q}(0)\right) \frac{1}{q_{\mathrm{scl}}} \\ \\ \left(\boldsymbol{u}(T) - \boldsymbol{u}(0)\right) \frac{1}{\Omega q_{\mathrm{scl}}} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} \frac{1}{a}\boldsymbol{q}_{-}(0) \\ \\ \frac{1}{a\Omega}\boldsymbol{u}_{-}(0) \\ \omega \\ \zeta \end{bmatrix}.$$

The amplitude and phase normalization is done by prescribing $q_{i_{\mathrm{norm}}}(0) = a$ and $u_{i_{\mathrm{norm}}}(0) = 0$. The vectors $\boldsymbol{q}_{-}(0)$ and $\boldsymbol{u}_{-}(0)$ are essentially $\boldsymbol{q}(0)$ and $\boldsymbol{u}(0)$ but without the above prescribed values.

A Newmark Integrator

In NLvib, the constant average acceleration variant of the Newmark-β method is implemented. Newmark methods are particularly popular for numerical time step integration in structural dynamics. The constant average acceleration variant is implicit and enjoys unconditional stability. It is implemented in most FE tools. We only summarize its implementation here, for details on its numerical stability and accuracy, and relations to other methods we refer to [1].

We assume a sequence of N equidistant time instants $\{t_k\}$ on the time interval $[0, T[$,

$$t_k = k\Delta t \quad k = 0, \ldots, N-1\,, \tag{C.9}$$

with $\Delta t = T/N$. The constant average acceleration method applies the trapezoidal quadrature rule to both $\boldsymbol{q}$ and $\boldsymbol{u}$,

$$\begin{aligned} \boldsymbol{u}_{k+1} &= \boldsymbol{u}_k + \frac{\dot{\boldsymbol{u}}_k + \dot{\boldsymbol{u}}_{k+1}}{2}\Delta t\,, \\ \boldsymbol{q}_{k+1} &= \boldsymbol{q}_k + \frac{\boldsymbol{u}_k + \boldsymbol{u}_{k+1}}{2}\Delta t\,. \end{aligned}$$

Herein, $\boldsymbol{q}_k$ denotes $\boldsymbol{q}(t_k)$ and so on. From this, we can follow

$$\dot{\boldsymbol{u}}_{k+1} = \frac{4}{\Delta t^2}\left(\boldsymbol{q}_{k+1} - \boldsymbol{q}_k\right) - \frac{4}{\Delta t}\boldsymbol{u}_k - \dot{\boldsymbol{u}}_k\,, \tag{C.10}$$

$$\boldsymbol{u}_{k+1} = \frac{2}{\Delta t}\left(\boldsymbol{q}_{k+1} - \boldsymbol{q}_k\right) - \boldsymbol{u}_k\,. \tag{C.11}$$

Consider the equation of motion evaluated at t_{k+1},

$$\boldsymbol{M}\dot{\boldsymbol{u}}_{k+1} + \boldsymbol{D}\boldsymbol{u}_{k+1} + \boldsymbol{K}\boldsymbol{q}_{k+1} + \boldsymbol{f}_{\mathrm{nl}}\left(\boldsymbol{q}_{k+1}, \boldsymbol{u}_{k+1}, t_{k+1}\right) - \boldsymbol{f}_{\mathrm{ex}}\left(t_{k+1}\right) = \boldsymbol{0}. \tag{C.12}$$

Substitution of Eqs. (C.10)–(C.11) into Eq. (C.12) yields an implicit equation in the displacement, $\boldsymbol{q}_{k+1}$, at the end of the time step,

$$\boldsymbol{r}\left(\boldsymbol{q}_{k+1}\right) = \boldsymbol{S}_{\mathrm{d}}\boldsymbol{q}_{k+1} + \boldsymbol{f}_{\mathrm{nl}}\left(\boldsymbol{q}_{k+1}, \boldsymbol{u}_{k+1}\left(\boldsymbol{q}_{k+1}\right), t_{k+1}\right) - \boldsymbol{b}_{\mathrm{d}} = \boldsymbol{0}\,, \tag{C.13}$$

where $\boldsymbol{u}_{k+1}\left(\boldsymbol{q}_{k+1}\right)$ is substituted from Eq. (C.11), and with

$$\boldsymbol{S}_{\mathrm{d}} = \frac{4}{\Delta t^2}\boldsymbol{M} + \frac{2}{\Delta t}\boldsymbol{D} + \boldsymbol{K}\,, \tag{C.14}$$

$$\boldsymbol{b}_{\mathrm{d}} = \boldsymbol{f}_{\mathrm{ex}}(t_{k+1}) + \boldsymbol{M}\left(\frac{4}{\Delta t^2}\boldsymbol{q}_k + \frac{4}{\Delta t}\boldsymbol{u}_k + \dot{\boldsymbol{u}}_k\right) + \boldsymbol{D}\left(\frac{2}{\Delta t}\boldsymbol{q}_k + \boldsymbol{u}_k\right)\,. \tag{C.15}$$

If the system is linear, so is the algebraic equation system (C.13). In the nonlinear case, the equation system is solved using Newton iterations with Cholesky factorization of the analytically determined Jacobian.

In NLvib, the Newmark method is actually implemented in normalized time $\tau = \Omega t$. The derivatives thus become

$$\boldsymbol{u} = \dot{\boldsymbol{q}} = \frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = \underbrace{\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}\tau}}_{\boldsymbol{q}'}\underbrace{\frac{\mathrm{d}\tau}{\mathrm{d}t}}_{\Omega} = \Omega\boldsymbol{q}'$$

$$\dot{\boldsymbol{u}} = \ddot{\boldsymbol{q}} = \ldots = \Omega^2\boldsymbol{q}''$$

The equations of motion in normalized time are

$$\underbrace{\boldsymbol{M}\Omega^2}_{\tilde{\boldsymbol{M}}}\,\boldsymbol{q}'' + \underbrace{\boldsymbol{D}\Omega}_{\tilde{\boldsymbol{D}}}\,\boldsymbol{q}' + \boldsymbol{K}\boldsymbol{q} + \boldsymbol{f}_{\mathrm{nl}}\left(\boldsymbol{q}, \Omega\boldsymbol{q}', \tau\right) - \boldsymbol{f}_{\mathrm{ex}}(\tau) = \boldsymbol{0}\,. \tag{C.16}$$

This is the point of departure for the application of the Newmark method in NLvib.

An Almost Foolproof Way to Analytical Gradients

Analytical derivatives commonly permit a substantial speedup in gradient-based numerical root-finding and optimization. This is especially true for large-dimensional problems and if the alternative is to use finite-difference approximations. On the other hand, developing them by hand is a rather tedious and dull endeavor. Particularly, if one does not have much experience with this yet, it is a very error-prone task, too, and can lead to a lot of frustration. The way, derivatives are analytically calculated in NLvib is rather simple, and it is explained in the following.

Attempt to learn from the example code:

```
function [R,dR] = my_function(X,param1,param2)
% Define auxiliary variables from input variables X
x1 = X(1);
x2 = X(2);
Om = X(3);

% Initialize derivative of auxiliary variables ('
   Seeding')
dX = eye(length(X));
dx1 = dX(1,:);
dx2 = dX(2,:);
dOm = dX(3,:);

% Operate on auxiliary variables, determine
   derivatives in each step using elementary calculus
z = x1*Om^2;
dz = dx1*Om^2 + x1*2*Om*dOm;
R = z/x2 - x2;
dR = dz/x2 - z/x2^2*dx2 - dx2;
```

Herein, X is a vector of real-valued variables and has here length 3. R is scalar in this example but can generally have arbitrary length. dR is the derivative matrix $[J_{ij}]$ with $J_{ij} = \partial R_i/\partial X_j$. In the first lines of the code, the elements of X are interpreted (stored as auxiliary variables). What is perhaps least common to the prospective readers is the *seeding* in the middle. Here, the elementary derivatives of the auxiliary variables are determined. This seems useless at first, but this is precisely what allows us to apply the chain rule in a straightforward way, whenever an operation involves quantities depending on (elements of) X. This is the case in the last four lines of the code.

It is the simplicity of the described procedure that, in our experience, avoids many errors. A disadvantage of the procedure is that it does not necessarily lead to the most efficient implementation. If one substitutes the derivatives into one another and carefully analyzes them, it might be possible to further optimize the code for a given computing architecture.

As mentioned earlier, *automatic differentiation* can be used to completely avoid the tedious manual calculations. Actually, the manual scheme described above is very closely related to automatic differentiation with operator overloading, where the chain rule is recursively applied in every line of the initial code to determine the derivative in addition to the function value during runtime.

Practical Tips on HB and Continuation

When implementing and starting to work with HB and continuation methods, one will almost certainly run into difficulties and make errors that were encountered by many other people before.

Strongly simplify your problem first and then successively increase complexity!
In other words, *live a homotopy*. Mind the following:

1. *Always analyze the linearized problem first!* In particular, check the following:
   - Do the system matrices have the expected dimensions, symmetries, eigenvalues?
   - Derive a suitable initial guess for the nonlinear analysis.
   - Derive typical values for diagonal preconditioning/linear scaling c.f. Sect. 4.3).
2. *Always start the nonlinear HB analysis with $H = 1$!*
3. *Then increase H*, until the results stabilize. Do not waste resources by setting it unreasonably high. The optimal value depends, among others, on the problem (degree of smoothness of the nonlinear terms, ...) and the quantities of interest (displacements, velocities, forces, ...).

What shall I do if I encounter one or more of the following difficulties during solution or continuation?

1. *Initial guess not within basin of attraction.* If you do not find a first solution point, do the following:
   - Start in a *more linear* regime.
   - Derive a better initial guess. Use a suitable linearization (appropriate boundary conditions), or a lower quality numerical approximation (lower harmonic truncation order, fewer number of degrees of freedom), or numerical integration, or a (semi-)analytical approximation (single-term HB, multiple scales, perturbation calculus, etc.).
   - If using analytical gradients, check if they are correct. If you run NLvib's continuation function with the jac parameter set to none, and the problem no longer occurs, your analytical gradients are likely to be wrong.

2. *No convergence during continuation.*
    - Use appropriate preconditioning. In particular, apply suitable linear scaling of the unknowns, perhaps also for the residual. In NLvib, you may use the Dscale vector to apply linear scaling to the unknowns.
    - Reduce (nominal) step length. In NLvib, reduce the parameter ds.
    - Ensure that numerical path continuation is activated. In NLvib, make sure that the continuation flag is set to 1 (default).
    - Increase the number of time samples per period within the AFT scheme.
    - If using analytical gradients, check if they are correct, c.f. above.
3. *The computation time is very large.*
    - Use appropriate preconditioning, see above.
    - Increase step length. In NLvib, increase the parameter ds.
    - Use (correct!) analytical gradients.
    - Learn more about efficient programming. In MATLAB, try to avoid loops by vectorization, use fast built-in functions where applicable, and use sparse storage where appropriate. In general, consider the computing architecture to parallelize your code and optimize the memory usage.
    - Lower your expectations!

The list of hints is of course not complete. In the world of numerical methods, practical experience is an important source of knowledge. We are grateful for receiving feedback on NLvib and suggestions for additional hints we can include in an updated version of this list on our website.

References

1. M. Géradin, D.J. Rixen, *Mechanical Vibrations: Theory and Application to Structural Dynamics*, (Wiley, New York, 2014)
2. M. Krack, Nonlinear modal analysis of nonconservative systems: extension of the periodic motion concept. Comput. Struct. **154**, 59–71 (2015). https://doi.org/10.1016/j.compstruc.2015.03.008