USER GUIDE MANLAB 2.0

Contacts:

Sami Karkar, karkar@lma.cnrs-mrs.fr

Bruno Cochelin, cochelin@lma.cnrs-mrs.fr

Christophe Vergez, vergez@lma.cnrs-mrs.fr

Olivier Thomas, olivier.thomas@cnam.fr

Arnaud Lazarus, lazarus@ladhyx.polytechnique.fr

October 4, 2011

Credits :

Version 2.0 editor : Sami Karkar

Version 1.0 editor : Rémy Arquier

Stability analysis : Arnaud Lazarus and Olivier Thomas (Hill method), Cyril Touzé (RK4)

Visualisation scripts : Olivier Thomas

Examples : Bruno Cochelin, Christophe Vergez, Sami Karkar, Olivier Thomas

Contents

0.1 WHAT'S NEW IN MANLAB 2.0?

Manlab 2.0 provides some improved and some new capabilities for path-following and bifurcation analysis, with focus on fixed points and periodic orbits of dynamical systems. New features include :

- improved graphical user interface
- improved continuation of periodic orbits through HBM using fortran acceleration
- stability analysis for both fixed points and periodic orbits
- bifurcation detection for both fixed points and periodic orbits
- bifurcation analysis : classical codimension 1 bifurcations are recognized (simple bifurcation, period doubling, Neimark-Sacker –also known as secondary Hopf)

Check out the new examples and see details of the new features in chapter 1 and 2.

Chapter 1

HOW TO USE MANLAB

1.1 What is Manlab

In many scientific areas, one wants to solve nonlinear algebraic systems of equations of the form [1]

$$R(U) = 0 \tag{1.1}$$

where R is a vector of n equations and U a vector of $n+1$ unknowns. When R is smooth, the solutions of (1.1) is made of one or several continuous branches. The drawing of these branches in a (U_i, U_j) plane is called a bifurcation diagram. Here, U_i and U_j designate two components of the vector U (see figure 1.1 for an illustration).

A classical strategy for solving (1.1) is to continue the branches of solutions from given solution points. This means to travel on a branch of solutions, to detect when another branch crosses (bifurcation) and, if desired, to switch to the new branch. This continuation process is also refered to as path following technique [1, 2] .

Manlab is a graphical interactive software for the continuation of branches of

[1] In scientific literature related to continuation, the system (1.1) is often written $R(U, \lambda) = 0$ where R is a system of n equations, $U \in \mathrm{R}^n$ a vector of unknowns and $\lambda \in \mathrm{R}$ a parameter

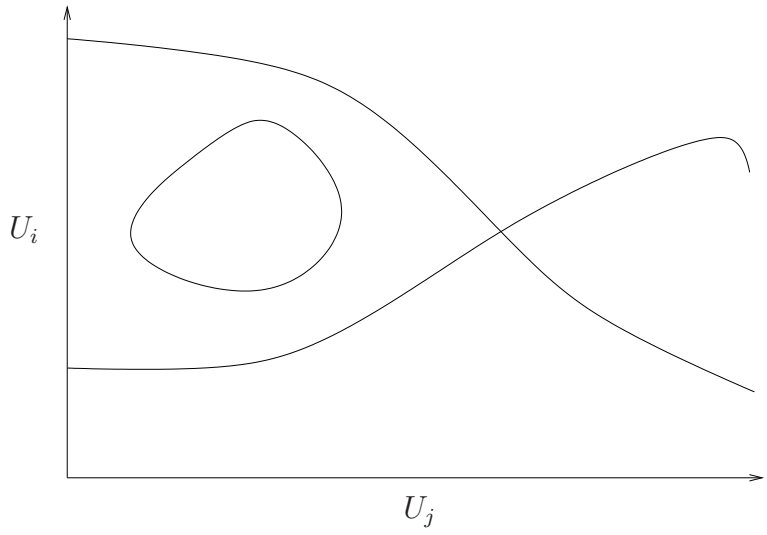


Figure 1.1: *Illustration of a bifurcation diagram showing various branches of solutions.*

solutions of system (1.1). Its solver is based on the Asymptotic Numerical Method (ANM in english, MAN in french). At each step of continuation, the branch of solutions is given by a power series expansion with respect to the pseudo-arc length parameter. By using a high order of truncature, a continuous acurate description of the solution branches is obtained. Because the series contain many usefull information, the continuation and the detection of bifurcation is very robust [3, 4].

In the context of non-linear dynamical systems analysis, **Manlab** can provide (since version 2.0) linear stability analysis of equilibrium points and periodic orbits (using the Harmonic Balance Method), together with automatic bifurcation detection and bifurcation type recognition. Thus, the user is now able to get comprehensive information about the physics of the system under study.

Manlab is an object-oriented **Matlab** program. Its graphical interface allows the interactive control of the continuation process: computation of a portion of a branch, choice of a new branch at a bifurcation point, reverse direction of continuation on the same branch, jump, visualization of user-defined quantities at a

particular solution point, selection and deletion of a branch, or of one of its portion. This set of functions proved to provide flexibility and efficiency during the continuation process.

From a practical point of view, the user has to define the problem to solve as a **Matlab** object, which contain the functions[2] allowing the calculation of the vector R of the system of equations. Thanks to the flexibility offered by the **Matlab** environment, users become rapidly familiar with **Manlab**. Calls to external routines (e.g. finite elements code) are possible.

1.2 PREREQUISITE

1.2.1 Theoretical background

Manlab can be used as a black box and requires no particular theoretical background. However, it may be helpfull to known the principle of continuation based on predictor-corrector technique. They are well decribed in classical textbooks [2],[5],[6].

1.2.2 Matlab prerequisite

Manlab is intended for version number of **Matlab** larger than (or equal to) $R2007b$ for Linux and MacOS systems, $R2009b$ for MS Windows systems. The user is supposed to be familiar with basic operations on vectors and structures in **Matlab**. Experience in object-oriented programming is not required.

To use the Fortran acceleration capability, a valid fortran compiler is needed, as well as the `mex` utility of **Matlab** (available to most distribution, but optional).

[2] so-called "methods" in object-oriented programming.

It must be compatible with the version of `mex` you are using : Windows users shall use `g95`, MacOS and Linux users should use `gfortran`.

1.3 HOW TO GIVE THE R VECTOR IN MANLAB

In **Manlab**, the branches of solutions are sought as power series expansion of a path parameter a :

$$U(a) = \sum_{i=0}^{N_{order}} a^i U^i \tag{1.2}$$

and the order of truncature N_{order} is generally high, between 15 and 30. For an easy and efficent computation of the power series, the vector of equation R has to be polynomial and quadratic. More precisely, Manlab deals with systems of the form :

$$R(U) = L0 + L(U) + Q(U, U) = 0 \tag{1.3}$$

where $L0$ is a constant vector, L a linear operator with respect to U, and Q a bi-linear operator with respect to U. This quadratic framework could appear as very restrictive at first. However, as we shall see along this manual, a very large class of algebraic systems can be put under that framwork provide that some transformations are performed and additional variables are added.

Let's take an example. We want to solve the following system

$$\begin{aligned} r_1(u_1, u_2, \lambda) = &\ 2u_1 - u_2 + 100\tfrac{u_1}{1+u_1+u_1^2} - \lambda &= 0 \\ r_2(u_1, u_2, \lambda) = &\ 2u_2 - u_1 + 100\tfrac{u_2}{1+u_2+u_2^2} - (\lambda + \mu) &= 0. \end{aligned} \tag{1.4}$$

Introducing the following additional variables $v_1 = u_1 + u_1^2$, $v_2 = u_2 + u_2^2$, $v_3 = \frac{1}{1+v_1}$

and $v_4 = \frac{1}{1+v_2}$, the system is now equivalentely rewritten as,

$$\begin{array}{llll} 0 & +2u_1 - u_2 - \lambda & +100u_1v_3 & = 0 \\ -\mu & +2u_2 - u_1 - \lambda & +100u_2v_4 & = 0 \\ 0 & +v_1 - u_1 & -u_1^2 & = 0 \\ 0 & +v_2 - u_2 & -u_2^2 & = 0 \\ -1 & +v_3 & +v_1v_3 & = 0 \\ \underbrace{-1}_{L0} & \underbrace{+v_4}_{L(U)} & + \underbrace{v_2v_4}_{Q(U,U)} & = 0 \end{array} \tag{1.5}$$

with $U = [u_1, u_2, v_1, v_2, v_3, v_4, \lambda]$.

To put a given system under the required formalism 1.3 is generally the most difficult and unusual task for the beginner with Manlab.

1.4 QUICK INSTALLATION

- Uncompress the archive `manlab_(version).tar.gz` in a directory. In the archive are gathered **Matlab** files required by **Manlab** together with various examples detailed hereafter. The tree structure of `manlab_(version).tar.gz` is detailed below:
  `MANLAB/SRC/` ← source files mandatory for the functionning of **Manlab**
  `MANLAB/DOC/` ← documentation
  `MANLAB/BASIC-EXAMPLES/` ← various examples ready to work with

- Add the path of directories `MANLAB/SRC/` to the `path` variable of **Matlab**.

  Example : if the archive has been unpacked in the directory `/home/myself/applications/`, you only have to type in the **Matlab** console the following commands:

  ```
  > addpath('/home/myself/applications/MANLAB/SRC');
  ```

Manlab is now installed on your computer.

Note : If you want to use fortran acceleration, you will need extended installation as described in chapter 2.

1.5 QUICK START

1.5.1 How to launch examples ?

Matlab scripts allowing to run the examples can be found in the directory `BASIC-EXAMPLES` of the archive. Make this directory your working directory in **Matlab**, and type in `parabole` in the command line to launch the parabole example. The graphical interface of **Manlab** is made visible and the continuation can be started using the button "+>".

1.5.2 Simple detailed example

To have a quick, but yet deep understanding of **Manlab**, the user is invited to read and execute the source codes of the example below. They demonstrate the continuation of a parabola. The unknow vector is $U = [x, y]^t$ and the vector of equation

$$R(U) = y - (x - a)^2 = 0 \tag{1.6}$$

$R(U)$ has to be rewritten as a 'LQ' problem:

$$R(U) = L0 + L(U) + Q(U, U) = 0 \tag{1.7}$$

$L0$, L, Q contain the constant, linear and quadratic terms respectively:

$$L0 = -a^2, \quad L(U) = y + 2ax, \quad Q(U1, U2) = -x_1 x_2 \tag{1.8}$$

with $U1 = [x_1, y_1]^t$ and $U2 = [x_2, y_2]^t$.

The files needed to run this example (`L0.m`, `L.m`, `Q.m` and `PARABOLE.m`) can be found in the directory `BASIC-EXAMPLES/@PARABOLE`. The first three files contain the definition of the functions $L0$, L et Q used to calculate the vector of equations R, and the file `PARABOLE.m` is the constructor (in the sense of object-oriented programing). Finally the instructions which allow to launch the graphical interface with the user defined parameters are written in the script file `parabole.m` located in the `BASIC-EXAMPLES` directory (same level as `@PARABOLE`).

1.5.3 Construction of the PARABOLE object

The file below allows to create an object containing data linked to the problem. In the present case of a parabola defined in Eq. 1.6, the unique data is the constant parameter a.

File : PARABOLE.m

```
function objparabole = PARABOLE(a)
    % creation of a structure containing the constant a
    structparabole.a = a;

    % creation of a manlab  objet which defines a system
    % with two unknowns and one equation (the number of
    % unknowns should be passed as an argument)
    objsys = SYS(2);

    % creation of the PARABOLE oject containing the data
    % of the structure structparabole and heriting from
    % the the object objsys
    objparabole = class(structparabole,'PARABOLE',objsys);
end
```

The command `objsys = SYS(2);` creates an object which is a nonlinear sys-

tem for **Manlab**. The command
`objparabole = class(structparabole,'PARABOLE',objsys);` allows to link the object `objsys` with the structure `structparabole` and returns the resulting object `objparabole` whose type is `PARABOLE`.

While these command lines may appear unclear to the reader unfamiliar with object-oriented programming, it should be noticed that these few lines allowing the object creation are almost always the same ! Only the name of the object (here `PARABOLE`) and the number of unknowns should be changed. This can be checked in the examples presented at the end of this userguide. Therefore, no need to understand subtleties of the object-oriented programming, working by analogy should work.

1.5.4 Functions L0, L, et Q

File : L0.m

```
function L0 = L0(objparabole)
    % L0 = -a^2
    L0 = -objparabole.a * objparabole.a;
end
```

File : L.m

```
function L = L(objparabole,U)
    % L = y + 2 a x
    L = U(2) + 2 * objparabole.a * U(1);
end
```

File : Q.m

```
function Q = Q(objparabole,U1,U2)
    % Q = - x1 * x2
    Q =  -  U1(1) * U2(1);
end
```

1.5.5 Launching script

The following script, located in the BASIC-EXAMPLE directory, allows to launch the continuation of a branch of parabola:

```
> manlabinit;            % initialisation of Manlab
> a         = 1;         % definition of the constant a
> ML_problem  = PARABOLE(a); % creation of a PARABOLE object
> ML_Ustart   = [ a; 0; ];   % definition of a vector containing
                         % an approximate solution
> ML_dispvars = [ 1, 2];     % definition of a vector containing
                         % the index of the variables to plot (x,y)
> manlabstart;           % launching of manlab
```

Once this script has been launched, the graphical interface of **Manlab** is made visible and the continuation can be started using the button “+>”.

1.6 A CLOSER LOOK

A more detailed and precise description of what has been presented above is given hereafter.

1.6.1 Classes and object-oriented programming within Matlab

In the following, some concepts allowing the writting of **Matlab** classes are reviewed. For more details, the reader should refer to the **Matlab** documentation.

From a pragmatical point of view, a **Matlab** class is defined by a set of source files in the same directory. The name of that directory must begin with the character @, while the rest of the name is considered to be the name of the class.

Example : If you want to create a class named `CIRCLE`, a directory `@CIRCLE` should first be created and the files defining the class stored inside this directory.

In each of these files is written a function which manipulates the data embedded into the class. Among the files, one should be the constructor of the object. The constructor must have the same name as the class, for example : `CIRCLE`.

A call to the constructor function usually returns an object of the corresponding class type. The remaining functions in the directory are used to manipulate this object by manipulating the structure of data embedded in the object.

Here is an example of a **Matlab** class which defines the concept of a circle :

File : CIRCLE.m (constructor)

```
function objcircle = CIRCLE(centrex, centrey, radius )
    % definition of the structure of data of a circle
    structcircle.cx = centrex;
    structcircle.cy = centrey;
    structcircle.r = radius;

    % declaration of the  class CIRCLE embedding as a member
    % the structure structcircle
    objcircle = class(structcircle,'CIRCLE');
end
```

Once the file has been placed into the directory `@CIRCLE`, the command

```
> mycircle = CIRCLE( -1, 3, 6);
```

allows to create a circle object the centre of which is the point $(-1, 3)$ and the radius is 6. Both information are stored in the variable `mycircle`. Of course, without any additional function, this object is useless. Here is a function to plot the circle in a **Matlab** figure :

File : Draw.m (method)

```
function [] = Draw(objcircle, npoints)
    cx = objcircle.cx;
    cy = objcircle.cy;
    r = objcircle.r;

    xs = cx + r * cos((0:npoints-1)/nbpoints * 2 * pi);
    ys = cy + r * sin((0:npoints-1)/nbpoints * 2 * pi);

    plot(xs,ys);
end
```

Once this function has been placed inside the directory `@CIRCLE`, the following commands

```
> mycercle = CIRCLE( -1, 3, 6);
> Draw(mycircle, 20);
```

plot the circle in a figure using 20 points for the drawing. It is worth noting that the first argument of the functions applying to the class (so-called methods) should be an object which type is `CIRCLE`. If it is not the case, the function cannot work within **Matlab** as a method, and an error will occur.

It is possible to write as many methods as needed to work on data embedded within the class. Here is another illustration of a method which allows to translate the circle:

File : Translate.m (method)

```
function objcircle = Translate(objcircle, tx, ty)
    objcircle.cx = objcircle.cx + tx;
    objcircle.cy = objcircle.cy + ty;
end
```

Once this file has been placed in the directory `@CIRCLE`, the following commands allow to plot the original and the translated circles on the same figure:

```
% construction of the circle
> mycircle = CIRCLE( -1, 3, 6);
% Translation of the circle and copy in variable mycircletranslated
> mycircletranslated = Translate(mycircle, 2, 2);
% Plot the original circle in a figure
> Draw(mycircle, 20);
% Do not erase  the plot before next plot
> hold on;
% Plot the translated circle
> Draw(mycircletranslated, 20);
```

1.6.2 Definition of the user system (type 'LQ')

To work with **Manlab**, the system of equations to solve has to be implemented has a **Matlab** class. This class will obviously contain its constructor (the method used to create the class) as well as the methods allowing the calculation of the residue of the system of equations (L0, L and Q).

Moreover, in order to "link" your particular class to the **Manlab** solver, your class should derive (only one command line) from an existing class of type "**Manlab** system". See figure (1.2) for an illustration.

Your class should contain at least four functions :

- the first one allows the creation of the object. A structure of data related to the problem will possibly be defined (constant values, vectors, matrix, ...):
  `function obj = YOURSYS(yourparameters)`
- the three remaining functions L0, L, et Q should respect the following rules concerning the syntax and type of arguments:
  `function [L0] = L0(instance_yourobj)`
  `function [L] = L(instance_yourobj, U)`
  `function [Q] = Q(instance_yourobj,  U1,  U2)`

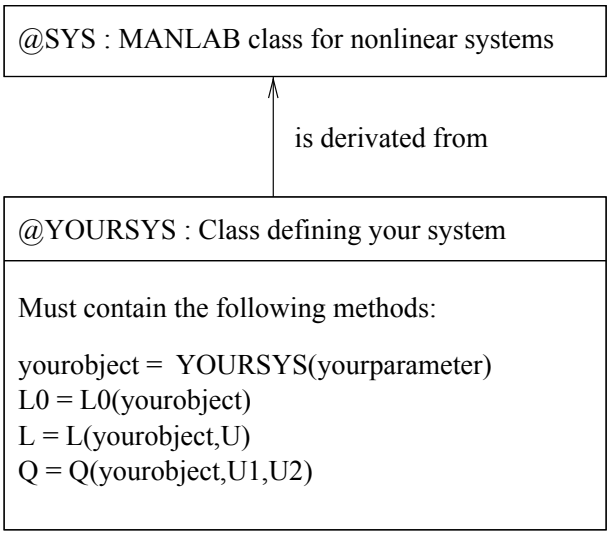


Figure 1.2: *Your class must contain the method required for its creation (the constructor) as well as the methods involved in the calculation of the residue. Moreover, your class should be derived from a more general class (type @SYS) given by* ***Manlab***

Each of these functions should be written in a separate file [3]. This is exemplified in figure (1.3).

Since release $R2006a$, the structure of class definition have changed, and methods may be declared in a unique file. However, the main structure of **Manlab** 2.0 still uses the old style for class definition. It might change in a future release.

Note that projects using HBM with fortran acceleration use a different parent class called `SYSHB`. The `SYSHB` class itself is derived from the `SYS` class. However, a few methods differs. Your object class should therefore comply with that specific class. Especially, the operators $L0$, L and Q methods have a different syntax, requiring additional arguments : `function [L0] = L0(instance_yourobj, H, Neq)`

[3] It is not possible to define more than one function in the same file, excepted for locally called functions (see **Matlab** documention)

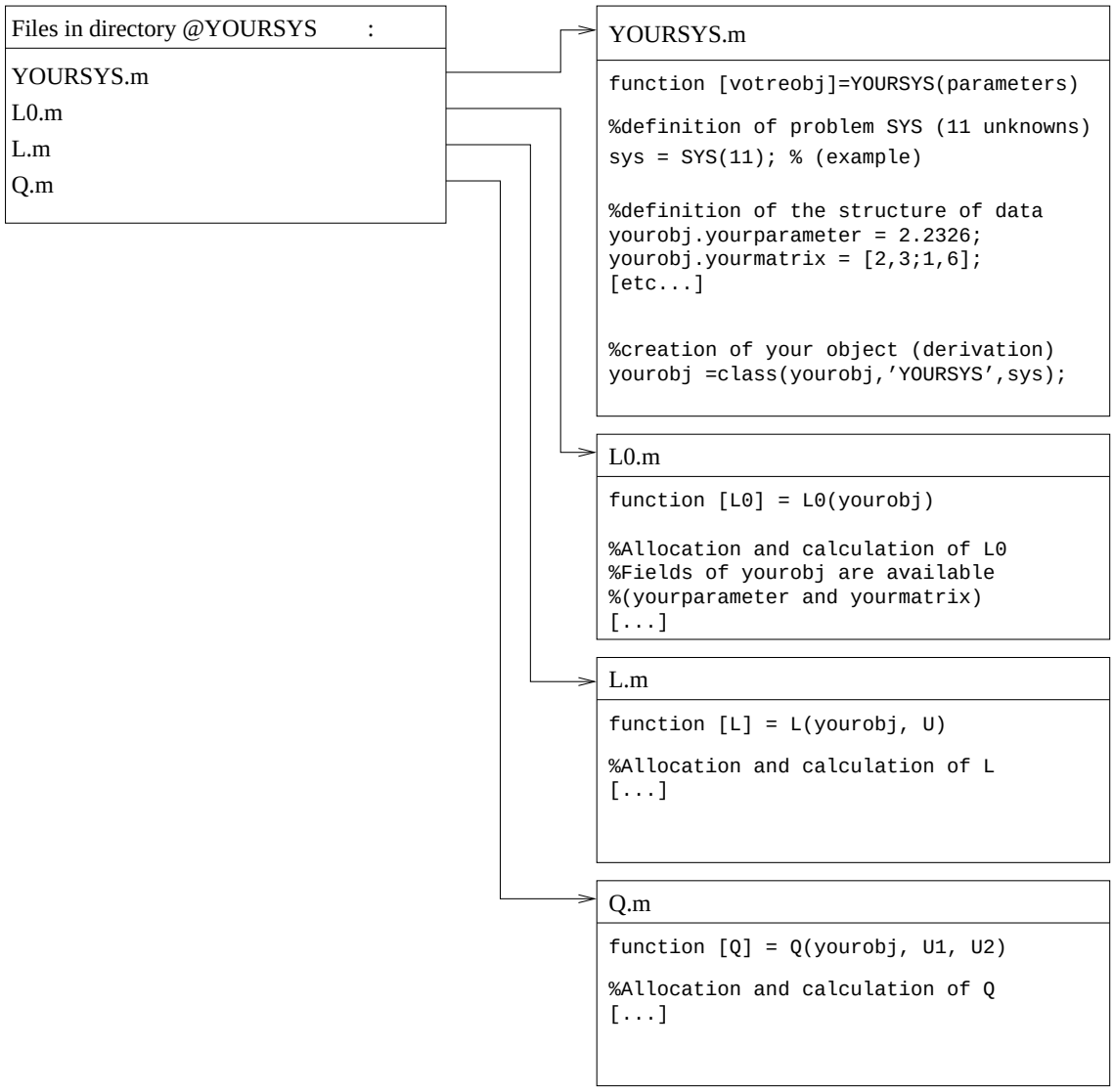


Figure 1.3: *Minimal example of the files organization defining a class named YOURSYS. There are four required functions which allow to define a 'LQ'-type quadratic problem. In the creator function YOURSYS, an object @SYS is created, as well as an object @YOURSYS, deriving from the class @SYS. Note that data embedded in your class (here* `yourmatrix` *and* `yourparameter`*) are available in all the functions located in directory @YOURSYS through the variable named* `yourobj`*.*

```
function [L] = L(instance_yourobj, U, H, Neq)
function [Q] = Q(instance_yourobj,  U1,  U2, H, Neq)
```

The inheritance command (last commands in the constructor) should then look like :

```
objsyshb = SYSHB(H,Neq,'forced');
                                    % or 'free'
instance_yourobj = class(instance_yourobj,'YOURSYS',objsyshb);
```

1.6.3 Launching

Launching **Manlab** is made through a script named `manlabstart` in the **Matlab** shell.
Variables `ML_objsys`, `ML_Ustart` and `ML_dispvars` must be predefined before the script is launched.

- Variable `ML_problem` must be be an object deriving from the class `SYS` (or `SYSHB` for fortran acceleration) and must embed functions `L0, L, Q` and a creator function (e.g. `YOURSYS`).
- Variable `ML_Ustart` is an approximate solution vector. Note that `ML_Ustart` must be a column vector (length `ninc`).
- Variable `ML_dispvars` is a matrix whose two columns are the index of the variables to plot in the bifurcation diagram. As an illustration:

$$\begin{bmatrix} x_1,\ y_1 \\ x_2,\ y_2 \\ x_3,\ y_3 \\ \vdots \\ x_n,\ y_n \end{bmatrix} \tag{1.9}$$

Please, note that the maximum number of simultaneous plots n is equal to 8. **Example** : If `dispvars=[1,2;1,4]` then two curves will be simultaneously plotted on the same diagram: `U(2)` as a function of `U(1)`, and `U(4)` as a function of `U(1)`. The number of curves to plot is determined automatically as being the number of lines in `dispvars`.

Example : Launching **Manlab** with an object whose type is YOURSYS, an approximate vector and a vector to precise which variables to plot.

File : lance.m

```
manlabinit; % Initialisation of Manlab
    % creation of an object whose type is YOURSYS
ML_problem  = YOURSYS(yourparameters);
    % definition of an approximate solution vector
ML_Ustart   =  [ 0; 2; 0.3; -2 ];
    % definition of a vector containing the index of the variables to plot
ML_dispvars = [ 1, 2];
    % call to the manlabstart script
manlabstart;
```

When the `manlabstart` is launched, **Manlab** tries to make a correction from the approximate solution `ML_Ustart` to come back right onto the solution. Once the correction step is over, the **Manlab** interface appears and you can start the continuation of solution branches of your problem. The starting point is indicated by a square box on the bifurcation diagram. A tangent vector at the starting point is also made visible on the diagram. Clicking the "+>" button make the continuation toward the direction indicate by the tangent vector. Clicking the "<-" button make the continuation in the reverse direction.

1.6.4 The class SYS

The class defined by the user must derive from the class SYS. Therefore, it can be seen as an interface between the user class and the rest of **Manlab** implementa-

tion.

In particular, the method for the creation of a SYS object is:

```
> objsys = SYS(ninc);
```

where `ninc` is the number of unknowns of the user problem. The number of equations is automatically set to `ninc-1`.

The class SYS returns an instance of an object (`objsys`) which allows to alter some **Manlab** parameters. These parameters can be modified through the following functions:

- `objsys = set_ordre(objsys, order)`. Allows to modify the order of the series expansion in the ANM process. The default value is 20.
- `objsys = set_itemax(objsys, itemax)`. Allows to modify the number of maximum iterations in Newton-Raphson correction. The default value is 15.
- `objsys = set_chemin(objsys, A)`. Allows to modify the path vector for the ANM. A should be a column vector with length $ninc$. By default, each component of A is set to 1.
- `objsys = set_nbptstroncon(objsys, order)`. Allows to modify the number of points in a section of a branch (used for the plot and the export of the section). The default value is 8.
- `objsys = set_nech(objsys,nech)`. Allows to modify the number of time samples used in the time-domain algorithm for stability analysis. Default value is 100.

An example of how to use these functions is given in section 4.2.3 page 65.

For details about the `SYSHB` class, please refer to chapter 2.

1.7 THE GRAPHICAL INTERFACE

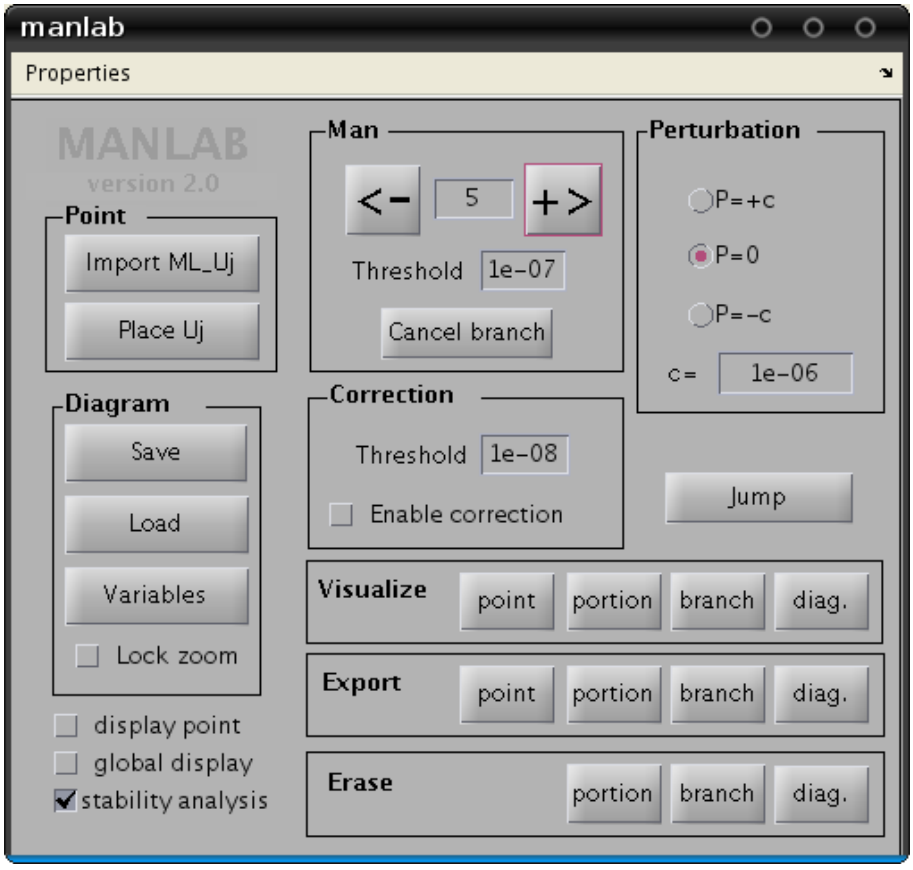


Figure 1.4: *Graphical interface of* ***Manlab***

1.7.1 The "Man" frame

+> (branch .) : Pressing this button launches the calculation of `Nb tron` sections from the current point. The direction of continuation is given by the tangent vector indicated on the bifurcation diagram.

<- (branch .) : Pressing this button launches the calculation of `Nb tron` sections from the current point. The direction of continuation is opposite to the tangent vector indicated on the bifurcation diagram.

Number '5' : Number of portions per branch. '5' is the default value
Threshold : Threshold relative to the residue in the ANM calculation
Cancel branch : Erase the latest branch calculated

1.7.2 Frame "Correction"

Threshold : Threshold relative to the correction.
Enable correction : If this box is checked, and if the starting point does not satisfy the tolerance criterion, a correction is applied until the norm of the residual vector become smaller than the threshold before starting the ANM calculation of the next portion

1.7.3 Frame "Perturbation"

P=+c : Add a "positive" perturbation to the original equations. The "intensity" of the perturbation can be controlled through its norm c. Note that a correction step is automatically launched when this button is pressed.
P=0 : No perturbation. However, a correction step is automatically launched when this button is pressed.
P=-c : Add a "negative" perturbation to the original equations. The "intensity" of the perturbation can be controlled through its norm c. Note that a correction step is automatically launched when this button is pressed.
c : Norm of the perturbation. Note that a correction step is automatically launched when the value of c is changed.

1.7.4 Frame "Visualize"

point : Launches the visualisation procedure `disp.m` (local plot) relative to a point in the diagram that you have to select with the mouse. The selected point is then displayed and become the current active point.
portion : Launches the visualisation procedure `disp_global.m` (global plot) relative to a section in the diagram that you have to select with the mouse.
branch : Launches the visualisation procedure `disp_global.m` (global plot) relative to a branch in the diagram that you have to select with the mouse.
diag. : Launches the visualisation procedure `disp_global.m` (global plot) for the complete diagram.

1.7.5 Frame "Export"

point : Export into the **Matlab** variable ML_Uj all the components of the vector U corresponding to the current active point. Additional information is returned if stability analysis is enabled.
portion : Export into the **Matlab** variable ML_Up all the components of the vector U for all points of a section, which should be selected with the mouse. The terms of the serie expansion of the selected branch are also exported in the **Matlab** variable ML_Ups. Additional information is returned if stability analysis is enabled.
branch : Export into the **Matlab** variable ML_Ub all the components of the vector U for all the points of a branch, which should be selected with the mouse. Additional information is returned if stability analysis is enabled.
diag. : Export all the components of vector U into the **Matlab** variable ML_Ud for all points of the diagram. Additional information is returned if stability analysis is enabled.

1.7.6 Frame “Erase”

portion, branch, diag. : Erase the element which will then be selected with the mouse in the main diagram window (or the whole diagram, in the last case).

1.7.7 Frame “Point”

Place Uj : Allows to select as a current active point, a point among branches already calculated. In practice, simply click with the mouse wherever wished on the diagram.
Import ML_Uj : The point specified in the global variable ML_Uj is chosen as the new current active point, and a correction is applied. This allows to place the current active point of **Manlab** through the **Matlab** shell. This function may be used if many approximate solutions are known, but are not connected by branches.

1.7.8 Frame "Diagram"

Lock zoom : Freeze/unfreeze the zoom scale of the main diagram plot (window : figure 2).
Load/Save : Allows to load a diagram previously saved. Note that a modification of the number of unknowns of a problem might cause an incompatibility with old diagrams.

1.7.9 The “Jump” button

This button allows to change the current active point through a jump procedure. This procedure can be seen as a first order predictor.

After clicking this button, click on the desired arrival area in the main diagram (windows : figure 2). The jump direction is given by the tangent to the branch at the current active point. The length of the jump is given by the distance from the current active point on the first curve (usually blue) to the selected arrival point.

Once the jump has been made, a correction step is automatically launched. If this correction step fails, the jump procedure is cancelled.

1.7.10 The “display point” check box

This check box allows to control the behaviour of **Manlab** concerning the local plots (see hereafter : USER-DEFINED PLOTS) without the command line. Its default value is related to the boolean-valued, global variable `ML_pointdisplay` that must be set before launching **Manlab** (1 : enabled, 0 : disabled).

1.7.11 The “global display” check box

This check box allows to control the behaviour of **Manlab** concerning the global plots (see hereafter : USER-DEFINED PLOTS) without the command line. Its default value is related to the boolean-valued, global variable `ML_globaldisplay` that must be set before launching **Manlab** (1 : enabled, 0 : disabled).

1.7.12 The “stability analysis” check box

This check box allows to enable or disable manually the linear stability analysis (see hereafter : STABILITY ANALYSIS) without the command line. Its default value is related to the boolean-valued, global variable `ML_stability` that must be set before launching **Manlab** (1 : enabled, 0 : disabled).

Note that you can disable/re-enable this functionality, only if it was enabled in the first place (at launch time). In that case, make sure you have defined proper

values for the global variable `ML_varstab` and that the functions needed for computing the jacobian matrix are available in your object class directory.

1.7.13 The “Properties” menu

In this menu, you can switch between the two algorithms of stability analysis, provided stability analysis has been enabled.

1.8 USER-DEFINED PLOTS

The continuation graph in **Manlab** only displays the plots of variables whose indexes are specified in variable `ML_dispvars`. To extend the plotting possibilities, two methods are available in **Manlab**:

1.8.1 Local user-defined plot

The local user-defined plot allows to plot whatever variable, or function of variables, at a given point of the continuation graph. To achieve this, a function `disp.m` should be created and placed in the directory of the class corresponding to the problem studied:

```
function [] = disp(instance_yourobj, Uj, Ujstab);
```

`instance_yourobj` is an instance of your object, `Uj` is a column vector containing all the variables of the problem, and `Ujstab` is an optional vector used for stability analysis (it must appear in the calling arguments sequence, whether or not it is used in the script). This function is called each time a continuation point is calculated and each time the button "plot point" is pressed.

Example : A very simple plot function

File : disp.m

```
function  [] = disp(obj, Uj, Ujstab)
    figure(3);
    bar(Uj); % bar plot of the values of all components of vector U
end
```

In the body of function `disp`, whatever **Matlab** command can be used (including various kinds of plots).

The default behaviour can be set in your launching script `lance.m` with the following command :

```
> ML_pointdisplay = 1     % local user-defined plots : 1=enable, 0=disable
```

When the checkbox `display point` is enabled, the `disp` function is called after each newly computed portion, as well as after a change of current active point using the “Place Uj” button.

The user can change the default behaviour at any time (except during a computation) by checking this box.

1.8.2 Global user-defined plot

The global user-defined plot relies on the same principle as the local one, excepted that it allows to reach all the points inside the portion of a branch. A function `disp_global` should be created and placed in the directory of the class corresponding to the problem studied:

```
function [] = disp_global(instance_yourobj, Us, Usstab);
```

`instance_yourobj` is an instance of your object, `Us` is a matrix whose column vectors contain all the variables of the problem, and `Usstab` is a matrix whose column vectors contain corresponding stability information. The number of columns depends on the number of points per portion `nbptstroncon`, which can be modified (see section 1.6.4).

The default behaviour can be set in your launching script `lance.m` with the following command :

```
> ML_globaldisplay = 1     % global user-defined plots : 1=enable, 0=disable
```

When the checkbox `global display` is enabled, the `disp_global` function is called after each new calculation of a portion, or when any of the buttons “Portion”, “Branch” or “Diagram” of the “Visualize” frame is used.

The user can change the default behaviour at any time (except during a computation) by checking this box.

1.8.3 Periodic orbits plotting scripts

A set of functions and scripts for drawing plots (local and global) for HBM periodic orbits, written by O. Thomas, are provided with **Manlab**. Here is a list of the functions available to users to easily write their own global displaying scripts.

General plotting settings :

`pencolorML.m`

Sets up 5 colours for fancier plots. Call this script first in your user-defined plot functions if you want to use these colours. It is already used by the other plotting functions.

Local plots :

`plotbarsincosHBM.m`

Plots a bar graph of all sine and cosine components amplitudes of the selected variables. The user must define the harmonic(s) number(s) and the variable(s) index(es) of interest.

`plotbarHBM.m`

Plots a bar graph of the harmonics amplitudes of the selected variables. The user must define the harmonic(s) number(s) and the variable(s) index(es) of interest, instead of the sin/cos amplitudes

`plotperiodHBM.m`

Plots a time-domain representation of variables. The user must define the index(es) of the variable(s) of interest.

`plotNNMHBM.m`

Plots a 3D surface representing the Nonlinear Normal Mode (invariant manifold) to which the current periodic orbit belongs. The user must provide indexes of the three variables of interest.

`plotfloquetmultHBM.m`

Plots in a convenient way Floquet multipliers of the current active point (there are $Ndof$ multipliers for a periodic solution). The figure is made up of three parts : on the right side, one can visualise Floquet multipliers in the complex plane and compare them with the unit circle ; on the left side, two small graphs represent the amplitude and the phase of Floquet multipliers as functions of the bifurcation parameter λ.
This function can be used both for local or global plots.

Global plots :

`plotbranchHBM.m`

Plots an amplitude diagram for the current portion or branch. The user must define the harmonic(s) number(s) and the variable(s) index(es) of interest.

`plotbranchampphaseHBM.m`

Plots an amplitude and phase diagram for the current portion or branch. The user must define the harmonic(s) number(s) and the variable(s) index(es) of interest.

`plotbranchbifHBM.m`

Plots an amplitude diagram for the current portion or branch, and mark the type of bifurcation (require stability analysis). The user must define the harmonic(s) number(s) and the variable(s) index(es) of interest, as well as the figure's number (avoid 2, which is the main diagram).

`plotfloquetmultHBM.m`

Plots in a convenient way Floquet multipliers of all analysed points (if the original physical system size is $Ndof$, there are $Ndof$ multipliers per point). The figure is made up of three parts : on the right side, one can visualise Floquet multipliers in the complex plane and compare them with the unit circle ; on the left side, two small graphs represent the amplitude and the phase of Floquet multipliers as functions of the bifurcation parameter λ.
This function can serve both for local or global plots.

For more details on any of these function, please type in the **Matlab** command line :

```
> help function_name
```

1.9 STABILITY ANALYSIS

Different algorithms are implemented in **Manlab** in order to analyse the linear stability of dynamical systems. If the check box "stability analysis" is enabled, then the linear stability analysis of solutions is performed along the computed branches. The algorithm used depends on the type of the solutions under study and on the selected algorithm : time-domain or frequency-domain.

Those algorithms rely on the computation of the jacobian matrix. Thus, some new functions (methods of your object) are needed to evaluate this matrix. The functions needed, as well as the algorithms used are detailed hereafter.

Note that users must :

- set `ML_stability=1` in their launching script in order to enable stability analysis. Then, they can temporary disable the feature, compute branches without it, re-enable the feature, etc...
- properly define the `ML_varstab` vector and the jacobian functions

1.9.1 Equilibrium point

If the user is studying equilibrium points (or fixed-points) of a dynamical system, both algorithms are equivalent. However, the user is reminded that each algorithm uses its own jacobian functions, as explain below.

In this case, the linear stability analysis consists in computing the eigenvalues ν_i (and eigenvectors) of the jacobian matrix at each point of analysis (there are `nbptstroncon` such points by portion). If any of the eigenvalues has a positive real part, then the current point (the associated periodic solution) is unstable. When following a branch that is, at first, stable, a bifurcation can be detected when one (or possibly more) of the eigenvalues crosses the imaginary axis.

1.9.2 Periodic orbits

If the user is studying periodic orbits of a dynamical system using the harmonic balance method (HBM), as shown in some of the examples, the frequency-domain method called *Hill's method* is recommended, but the user can also use the time-domain method called *monodromy matrix method*.

The default choice for the method used for computing the stability analysis is Hill's method, but can be overcome using the global variable `ML_algostab` in your launching script :

- `ML_algostab=1` selects the monodromy matrix method (time-domain algorithm)

- `ML_algostab=2` selects Hill's method (frequency-domain algorithm)

1.9.3 Methods used in the stability analysis

We describe here the two algorithm used for analysing the linear stability of fixed-points and periodic orbits, together with the additional functions needed in your object class directory.

Time-domain algorithm The time-domain method integrates the jacobian matrix over one period to get the monodromy matrix. The integration is carried out using a Runge-Kutta 4 algorithm on `nech` time samples. This sampling parameter can be modified using the `set_nech` method that your object class has inherited from the SYS class (see section 1.6.4 page 24).

The monodromy matrix method uses an additional function that returns the jacobian matrix of the physical system called `JT.m` :

File : JT.m

```
function [DT] = JT(instance_yourobject, Ustab)

    DT = [ JT_1_1 , JT_1_2,  ...     ;
           ...                       ;
           ...    , ..., JT_Ndof_Ndof ] ;
end
```

$Ndof$ is the size of the original physical system (before quadratic recast and HBM transformation) and $Ustab$, defined a little further, contains the values of the needed variables and lambda.

Frequency-domain algorithm Hill's method is fully described in [7]. It uses the three additional functions `J0.m`, `JL.m` and `JQ.m` that must be placed in your object class directory. They are defined as a decomposition of the jacobian matrix J so that $J0$ is a constant matrix, JL is a linear operator and JQ a quadratic operator on the variables given in $Ustab$[4] : $J = J0 + JL + JQ$.

File : J0.m

```
function [D0] = J0(instance_yourobj, Ustab)
    D0 = [ J0_1_1 , J0_2_1, ... ;
           ...                  ;
           ... , ..., J0_Ndof_Ndof ] ;
end
```

Only `Ustab(end)`, which contains lambda, may be used in this function.

File : JL.m

```
function [DL] = JL(instance_yourobj, Ustab)
    DL = [ JL_1_1 , JL_2_1, ... ;
           ...                  ;
           ... , ..., JL_Ndof_Ndof ] ;
end
```

All components of `Ustab` may be used in this function.

File : JQ.m

```
function [DQ] = JQ(instance_yourobj, Ustab, Vstab)
    DQ = [ JQ_1_1 , JQ_2_1, ... ;
           ...                  ;
           ... , ..., JQ_Ndof_Ndof ] ;
end
```

All components of `Ustab` and `Vstab` may be used in this function.

[4] lambda and omega are not considered as "variables" in the context of the jacobian matrix

1.9.4 Additional variables and parameters

The "Ustab" vector :

`Ustab` is a vector containing values of variables as well as the parameter λ, at the current analysis point. Only the variables needed for the computation of the jacobian matrix appear in this vector.

The indexes of these variables must be declared, in your launching script, in the global variable `ML_varstab` using the following command :

```
ML_varstab = [  var1_index, var2_index, ..., lambda_index, omega_index, H ];
```

The "tolstab" parameter :

In order to improve the quality of the detection of bifurcation, a tolerance threshold has been set on the stability test : a periodic solution will be declared unstable if at least one of its Floquet multipliers modulus exceeds $1 + tolstab$.

The default value for `tolstab` can be set through the global variable `ML_tolstab` in your launching script. For example :

```
ML_tolstab = 1e-3;
```

The automatic default value is $1.0\ 10^{-4}$.

Please note that :

- variables indexes refer to their indexes in the "small" system state vector, before HBM transformation (of size Neq)
- lambda and omega indexes refer to their indexes in the final vector U (of size $Neq + 1$ for fixed-points, or $Neq(2H + 1) + 1$ or $+2$ for periodic orbits of forced and free systems)

- The given value of H is the order of Fourier series truncation in Hill's method. It must be inferior or equal to the one used for the continuation. It must be equal to 0 for equilibrium points.

This additional global variable definition might appear awkward, but evaluating the jacobian matrix generally requires only a few variables and this way of proceeding will save a lot of computation-time, as compared to using the whole U vector.

For more detailed information about Hill's method, please refer to the article of Lazarus et al. [7].

Chapter 2

EXTENDED FEATURES : FORTRAN ACCELERATION

In order to improve **Manlab**, as the continuation of periodic orbits using the Harmonic Balance Method leads to very large number of equations, an acceleration strategy using Fortran is available.

2.1 INSTALLATION

2.1.1 Unix/Linux

Requirements :

Matlab releases $R2008b$ and later have been successfully tested.

Fortran acceleration requires the MEX compilation utility available in MATLAB (depending on your distribution), and a recent version of GFORTRAN (v4.0 or later is recommended). G95 have been reported to work, however some compilation options might need to be changed in the `Makefile`.

Setup :

- Run `mex -setup` in the **Matlab** command line and then choose the 'GNU compiler' option.
- Check the generated options file named `mexopts.sh` : it will have written `g95` instead of `gfortran` (Mathworks developpers seem to ignore that g95 is not the GNU fortran compiler), so make sure you correct this mistake first[1]
- Copy the file `MANLAB/TEMPLATES/Makefile.unix` to your class directory and rename it `Makefile`
- Edit this file and check for correct paths to your **Manlab** and **Matlab** directories

For more details about configuring the MEX utility, please refer to MATLAB documentation.

2.1.2 MacOS

Matlab releases $R2009b$ and later have a MEX utility compatible with GFORTRAN (v4.3 and later). Release $R2008b$ have been reported to work like on Unix/Linux systems.

The setup is the same as for Unix/Linux systems.

Some users reported a confusion between the `mex` command provided by **Matlab** and some other tools. Please make sure you provide the full path to the mex command in the `Makefile` you use.

[1] If you are lost and didn't succeed then, open a terminal and try the command `locate mexopts.sh` to locate the options file, then type `sed -i 's/g95/gfortran/g' /my/path/to/mexopts.sh`. This should do the job.
If you are lost even more, ask for your favorite geek to help you.

2.1.3 Windows

The use of Fortran acceleration is a bit more complex because of some obscure upper-case details.

Requirements :

Matlab release $R2009b$ has been successfully tested. Later releases should work as well.

It use to work on previous releases, however, some option names for MEX have changed since and we provide only valid `Makefile` for $R2009b$ and later.

Make sure you have installed the following (free and open-source) softwares :

- Mingw (http://www.mingw.org/) –tested release : 20100909 (contains GCC 4.5.1)
- g95-mingw –tested release : g95 v0.91
- gnumex (http://gnumex.sourceforge.net/) –tested release : 2.01

Note that, until now, no other compiler than G95 has been working for building MEX files under windows. In particular, we were unable to tackle the upper-case problem with GFORTRAN.

Setup :

The setup is a bit more complicated but shouldn't be too much of a challenge.

- **Step 1** :
  In the **Matlab** command line, get in the GNUMEX directory and start the GNUMEX utility by typing `gnumex`.

Check the path to your `MinGW` root directory and to your `MinGW\bin` directory for `g95.exe`.

Choose 'Fortran' option with the 'g95' compiler, language version '95'. Optimisation level 'O3' is recommended.

Build the option file, it should be named 'mexopts.bat' and placed in the correct directory by default.

- **Step 2** :
  Copy the template `Makefile.win` provided in `MANLAB\TEMPLATES` to your class directory and rename it `Makefile`.

  Edit this file and change the paths to your **Matlab** and **Manlab** locations.

  Make sure the path to the Fortran sources correspond to your problem type :
  free=autonomous forced=non-autonomous
  ———— CAUTION ————
  If you do not uncomment one of these two lines, an error will occur during compilation, saying the include file `fintrf_new.h` cannot be found.

  Save your Makefile.

- **Step 3** :
  Copy the `YOURPATH\TO\MinGW\bin\mingw32-make.exe` utility to your `C:\WINDOWS\SYSTEM32` directory and rename it `make.exe`.

  In your launching script `lance.m`, the compilation command should be similar to the following :

```
[status,result]=dos('cd @MYCLASS && make && cd ..','-echo');
```

They should be written before all other commands except the 'addpath' command.

That's it !

In case you want to remove all the temporary files and the compiled files (in order to disable Fortran acceleration, or after an upgrade on your equation files), use `make clean` instead of `make` in the commands above.

2.2 WRITING YOUR EQUATIONS IN FORTRAN

You will need to write only one file in Fortran language (preferably Fortran 95), named `petits_operateurs.f90` and located in the class directory of your project (ie. `@MYCLASS`).

This file must contain a module named `petits_operateurs` with :

- the values of some parameters of your system. It enables you to define once and for all the constants of the system that you can then use in your equations.
- the functions $pC0$, $pC1$, $pL0$, $pL1$, pQ and pM that correspond to your system's equations. It should look very much like the functions written in `L0.m`, `L.m` and `Q.m` except for fortran-specific syntax, thus it is usually very little work to translate the functions from the **Matlab** programming language to Fortran.

If you wish to use the stability analysis with Hill's method, you can also accelerate the computation by writing your own `jacobien.f90` file.

Check the examples provided using the HB demos (e.g. the forced Duffing oscillator described in [7]) and make sure you use the same header (especially the

`#include` command) as well as a good Fortran syntax. A good way to proceed is to copy an existing file from an example and then to edit only the parameters and the functions.

To test your code, go into the class directory in a terminal/console/command line and type 'make'. It should try to compile your code together with the Fortran sources of MANLAB-HB. In case error messages appear, try to debug your file from the first error statement (as errors usually propagate and give multiple error messages).

When your code compiles correctly, three new files will have been created : `L0.mexXXX`, `L.mexXXX` and `Q.mexXXX`. The extension is platform-dependant, hence `XXX` might be `glx`, `w32`, or another one depending on you CPU type and OS. To test the MEX-compiled operator functions, use the MATLAB command line and execute manually the first lines in your launching script `lance.m` where 'ML_problem' and 'ML_Ustart' are defined. Then try :

```
>> R=get_R(ML_problem, ML_Ustart); norm(R)
```

It should work and return the norm of the residue for the given starting point.

2.3 GENERAL NOTES ON THE USE OF FORTRAN WITH MANLAB

2.3.1 Parameters

Because it is painful to code and because the more exchanges between a Fortran routine and **Matlab**, the slower it gets, the Fortran routines currently does not get any of your object structure parameters.

Only H and Neq are automatically passed as explicit input arguments, the object itself is a dummy argument. Thus, if one would ever want to change any of the

parameters used in the equations, one should do so by editing the source code in `petits_operateurs.f90` itself.

It is a huge difference with most non-Fortran examples where the parameters of the system are defined in the 'lance.m' script and then passed to the class-constructor of the problem on the command :

```
>> ML_problem=MYOBJECT(H,Neq,param1,param2,...)
```

In a project using Fortran acceleration, changing the value of parameters only in the launching script will NOT (unfortunately) affect the Fortran source code, and thus the final computation.

2.3.2 Non-autonomous systems

As for now, the continuation of periodic orbit for non-autonomous systems is only available for a given (constant) amplitude F of the source term. The angular frequency ω is taken as the continuation parameter.

As a default, the source term for a periodically driven system has been written in the following form : $Fcos(\omega t)$.

Thus, the fortran source file `petits_operateurs.f90` should define a variable 'F' and its value.

Note also that the source term acts only on the first equation. If you want to use the source term on a different coordinate of your system (say number 4, for instance), you will need to edit the temporary Fortran file `L0.F90` and change $Neq+1$ to $Neq+4$ at the "forcing term" line (currently containing `L0(Neq+1) = F`), and then compile again manually using a second `make`.

2.4 A CLOSER LOOK ON "MAKE"

The `make` command –it is not a **Matlab** command, but a shell command– needs a `Makefile` in the directory it is being run. Makefiles are platform dependant, and require to be checked for correct paths and command names.

The provided Makefiles are designed to do two things when you execute 'make' :

1. New versions of fortran source files are checked and then, if needed, the template source files from **Manlab** are copied to your class directory : `L0.F90`, `L.F90`, `Q.F90` and `Hill_mat.F90`.
2. If any of the source files are newer than the compiled ones, or if compiled files are missing, compilation is done using the MEX utility until all compiled files are up-to-date.

Both steps can be performed manually, and for each of the three operators L0, L or Q, using the commands :

- `make X.F90` to copy the template file X.F90 from MANLAB-HB
- `make X.mex*` to compile the X.F90 and the using the MEX tool

To clean all compiled and temporary files use the shell command `make clean`.

To clean only the temporary files use the shell command `make cleantmp`.

You can modify the optimisation level of the fortran compiler by changing '-O3' to '-O2' or '-O'.

A simple way to compile without optimisation and with debugging symbols is to replace this with '-g' which is the debug flag. This can be useful if you need to debug fully your mex files, but chances are you will never need to do such an unpleasant work...

Note that all shell commands may be run from the **Matlab** command line by using the prefix `!`. The directory in which the shell command is run is the current working directory of **Matlab**.

2.5 Debugging Fortran MEX-files

If there is a bug, despite a correct compilation, MATLAB is likely to crash (sometimes without warning!), and you might need a specific debugging tool for Fortran. Please refer to **Matlab** documentation about MEX files written in Fortran.

Notice that most problems occur when the `ML_Ustart` variable does not have the right size (because the size is not checked in the Fortran routines and this might result in a segmentation fault), or when the system size Neq and harmonic truncation H are not consistent in **Matlab** scripts and Fortan source code.

If you think there is a bug in the Fortran source code provided within **Manlab**, please send a bug report to the first contact address. We will be happy to correct any of those.

Chapter 3

Theoretical elements

3.1 CONTINUATION

ANM is a continuation method to solve quadratic algebraïc nonlinear systems of smooth equations:

$$\mathbf{R}(\mathbf{U}) = 0 \qquad \text{with } \mathbf{R} \in \mathrm{R}^n \text{ and } \mathbf{U} \in \mathrm{R}^{n+1} \text{ is the vector of unknowns} \tag{3.1}$$

One specificity of the ANM is to give access to solution branches, and not only to solution points. To achieve this result, the vector of unknowns is written as a power serie expansion (truncated at order m) of a variable a (the so-called path parameter):

$$\mathbf{U}(a) = \Sigma_{i=0}^{m} \mathbf{U}_i a^i, \qquad \text{where } \mathbf{U}_i \in \mathrm{R}^n,\ a \in \mathrm{R}^+, \tag{3.2}$$

To change the truncation order m, see section 1.6.4 page 24.

More precisely, we only consider quadratic systems $\mathbf{R}$:

$$\mathbf{R} \triangleq \mathbf{L_0} + \mathbf{L}(\mathbf{U}) + \mathbf{Q}(\mathbf{U}, \mathbf{U}) \qquad \text{with } \mathbf{L_0}, \mathbf{L}, \mathbf{Q} \in \mathrm{R}^n \tag{3.3}$$

where $\mathbf{L_0}$ is a constant vector, $\mathbf{L}$ is a linear application, and $\mathbf{Q}$ is a bilinear form. The ansazt (3.2) is used in equation (3.3), leading to a polynomial expression. Equation

(3.1) is then reduced to term-to-term equalization of each power of a, which leads to a set of $m+1$ linear systems (since only the first $m+1$ powers of a are considered, from 0 to m).

Let $\mathbf{U}_0$ be a solution vector :

- **order 0 :** $\mathbf{L_0} + \mathbf{L}(\mathbf{U}_0) + \mathbf{Q}(\mathbf{U}_0, \mathbf{U}_0) = 0$, which is a trivial system since $\mathbf{U}_0$ is solution of (3.3).
- **order 1 :** $\mathbf{L}(\mathbf{U}_1) + \mathbf{Q}(\mathbf{U}_0, \mathbf{U}_1) + \mathbf{Q}(\mathbf{U}_1, \mathbf{U}_0) = 0$, this system can be rewritten $\mathbf{J}_{\mathbf{U}_0}\mathbf{U}_1 = 0$ where $\mathbf{J}_{\mathbf{U}_0} \in \mathrm{R}^{n \times n+1}$ is the jacobian matrix of $\mathbf{R}$ evaluated at $\mathbf{U}_0$.
- **order p>1 :** $\mathbf{J}_{\mathbf{U}_0}\mathbf{U}_p + \Sigma_{i=1}^{p-1}\mathbf{Q}(\mathbf{U}_i, \mathbf{U}_{p-i}) = 0$

The original nonlinear problem has then been reduced to a set of linear systems. However, the path-parameter a has still to be defined. Following Cochelin et al. ([3]), the classical definition of the pseudo arclength $a \triangleq < \mathbf{U}, \mathbf{U}_1 >$ is generalized to:

$$a \triangleq \mathbf{U}_1^t \mathbf{A} \mathbf{U}, \tag{3.4}$$

where $\mathbf{A}$ is a diagonal matrix, which allows to consider only some components of $\mathbf{U}$ for the definition of a. Once again, the use of the serie expansion (3.2) in (3.4), and the term-to-term equalization for each power of a leads to;

- **order 1 :** $\mathbf{U}_1^t A \mathbf{U}_1 = 1$
- **order p>1 :** $\mathbf{U}_1^t A \mathbf{U}_p = 0$

To change the values of $\mathbf{A}$, see section 1.6.4 page 24.

Remarks:

- Once each $\mathbf{U}_p$ has been found, the validity domain of the serie expansion is estimated through the calculation of an approximation of the convergence

radius a_{max} of the serie, i.e. the interval $[0\ a_{max}]$ for which $||\mathbf{R}(\mathbf{U}(a))|| \leq \epsilon_r$. The user-defined threshold ϵ_r is set through the graphical interface.

- The power serie expansion, and the convergence radius a_{max} define a portion of the branch. La décomposition en série, et son rayon de convergence. A new portion, continuating the branch, can be determined if the final point of the latest calculated portion is chosen as the starting point of the new portion (possibly after a correction step). A solution branch is therefore found by the ANM as a succession of portions, the length of which is automatically determined (through the estimation of the convergence radius of each power serie).

- **Correction step:** For reasons that wil be given later on, it may be useful to launch a correction step. It brings the starting point right back on the solution. For example, this allows to avoid cumulative errors when calculating a branch with many portions.

  For correction, a Newton-Raphson method is implemented in **Manlab** because of its quadratic convergence rate (since the correction is performed from solutions which are genrally close to the exact branch).

  Starting from an approximate solution $\mathbf{U_{ap}}$, the correction $\mathbf{\Delta U_{cor}}$ is seeked such that $\mathbf{R}(\mathbf{U_{ap}} + \mathbf{\Delta U_{cor}}) = 0$. This correction is obtained according to the Newton-Raphson algorithm through an iterative process:

$$\mathbf{\Delta U}_{i+1} = -\mathbf{J}^{-1}_{\mathbf{U_{ap}}+\mathbf{\Delta U}_i}\mathbf{R}(\mathbf{U_{ap}} + \mathbf{\Delta U}_i) \tag{3.5}$$

$$\mathbf{\Delta U_{cor}} = \mathbf{\Delta U}_{i+1} \text{ when } \mathbf{\Delta U}_{i+1} - \mathbf{\Delta U}_i \leq \epsilon_c \text{ where } \epsilon_c \text{ is a user-defined threshold} \tag{3.6}$$

  Since $\mathbf{\Delta U_{cor}} \in \mathrm{R}^{n+1}$ and (3.5) is a system of n equations, an additional equation is required. We chose to impose in **Manlab**:

$$\mathbf{U_t}^t \mathbf{\Delta U}_i = 0\ \forall i \text{ where } \mathbf{U_t} \text{ is defined by } \mathbf{J_{U_{ap}}}\mathbf{U_t} = 0 \tag{3.7}$$

This corresponds to impose that the direction followed by the correction algorithm to get closer to the branch is perpendicular to the branch.

Avantages of the ANM:

- The solution branch is known analatytically for each portion.
- Robustness of the method compared to other methods of continuation.
- Low computational cost.
- The quadratic nature of the equations makes the exact calculation of the tangent matrix an easy task. [1].

3.2 BRANCH SWITCHING THROUGH PERTURBATION

A classical strategy for the switch of branch at a bifurcation consists in slightly modifying the original equations by the addition of a low-norm perturbation vector. This transform the exact bifurcations into a perturbed bifurcations (see figure 3.1). The choise of the amplitude of the perturbation can be difficul with classical predictor-corretor schemes. The amplitude should be small enough to stay closed to the solution of the unpertutbed system, but not too small to avoid a systematic jump over the bifurcation. Du to its adaptative stepsize the ANM procedure is robust and succesfull almost whatver th amplitude of the perturbation.

In **Manlab**, the perturbation vector is a random vector, the norm and the sign of which are controlled through the graphical interface. This vector is used by the

[1] It is calculted by **Manlab** using L and Q.

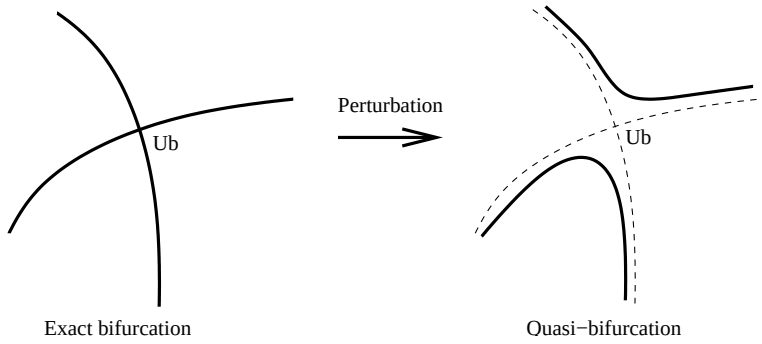


Figure 3.1: *Transformation of an exact bifurcation into a perturbed bifurcation through the addition of a small perturbation.*

@SYS class in a transparent way, and is inserted in the original problem as shown below:

$$\begin{aligned} R_p(U) &= R(U) + cP \\ &= L0 + L(U) + Q(U,U) + cP \end{aligned} \tag{3.8}$$

where $R_p(U)$ is the perturbed problem, P a normalized vector of constant random numbers, and c the intensity of the perturbation. If $R(U)$ has an exact bifurcation at a point U_b, it corresponds graphically to a crossing of two solution branches. The addition of a perturbation transforms the crossing into two a non crossing where the two branches remain separate[2] in the neighbourhood of U_b[5]. The distance between branches of the original and the perturbated problem depends on the intensity of the perturbation c. Moreover, changing the sign of c allows to have a symmetrical quasi-bifurcation. It is then possible to use the two perturbated plots to turn "left" or "right" at a (quasi-)bifurcation point, as shown in figure 3.2.

Finally, in order to go from the original to the perturbed problem and vice versa, a correction step is mandatory. This correction step is automatically launched by **Manlab** when needed. It uses a Newton-Raphson algorithm. The combination of different (positive and negative) values of the intensity of the perturbation and

[2]The method of the additional random vector, though not infallible (the crossing might remain), proves to work very well with a probability of failure close to zero.

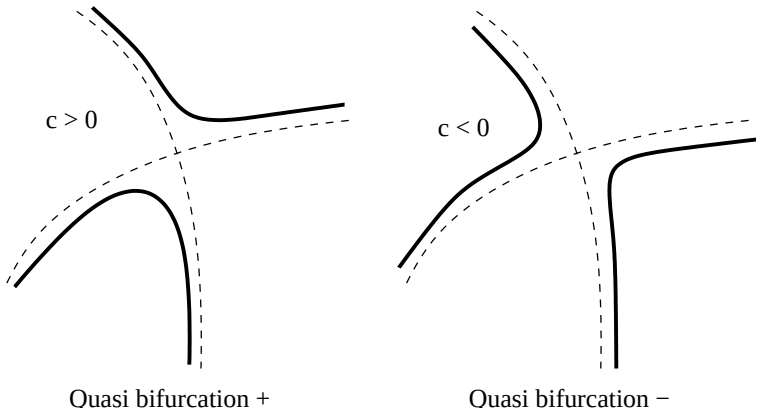


Figure 3.2: *Influence of the sign of c on the quasi-bifurcation.*

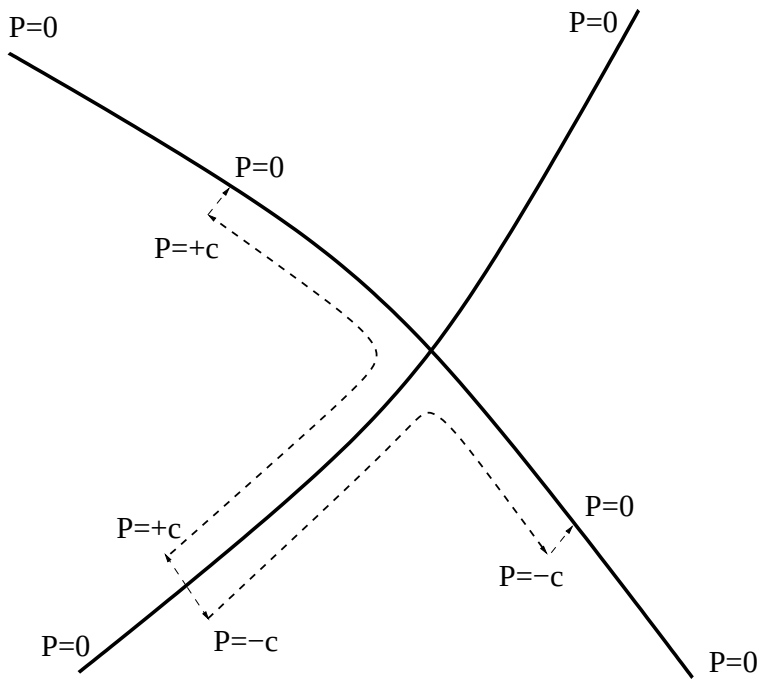


Figure 3.3: *Branch switching through a perturbation method*

several correction steps, **Manlab** allows the user to plot the different branches starting from a bifurcation point, as explained in figure 3.3.

Chapter 4

Simple Examples

4.1 THE FIRST EXAMPLE : "QUADMINI"

The following examples can be found in `MANLAB/BASIC-EXAMPLES/`.

4.1.1 Problem statement

Lets consider the following quadratic problem with a single equation and two unknowns:

$$R(U) = R([x, y]^T) = a + bx + cy + dxy + ex^2 + fy^2 \tag{4.1}$$

This problem can be recasted as

$$R(U) = L0 + L(U) + Q(U, U) \tag{4.2}$$

with

$$\begin{cases} L0 & = a \\ L(U) & = bx + cy \\ Q(U_1, U_2) & = dx_1y_2 + ex_1x_2 + fy_1y_2 \end{cases} \tag{4.3}$$

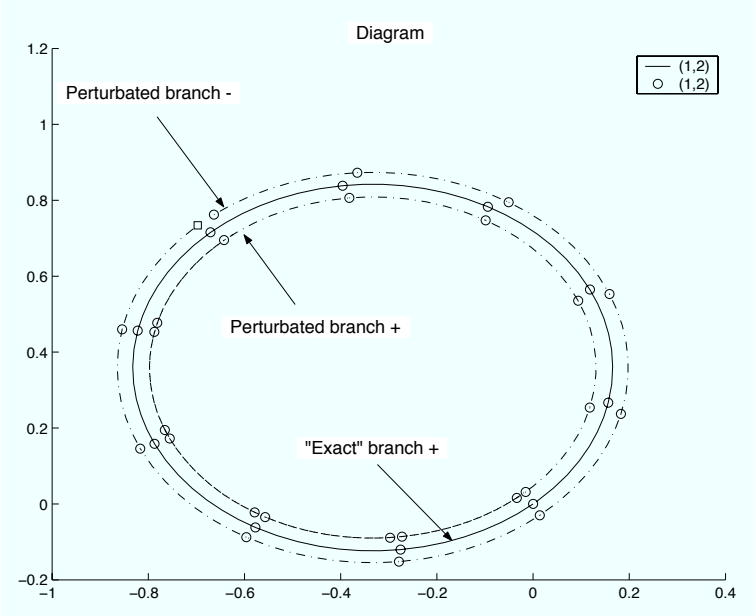


Figure 4.1: *Diagram of the QUADMINI problem when* $a = 0$, $b = 2$, $c = -2.3$, $d = 1$, $e = 2$, *and* $f = 3.2$. *The diagram shows the "exact" branch (a circle), as well as two perturbed branches (noted* $+$ *and* $-$*) when the intensity of the perturbation is* $c = 1e - 1$.

4.1.2 Definition of the user problem

Here are 4 files allowing to perform the continuation of the solutions with **Manlab**

File : QUADMINI.m

```
function obj = QUADMINI(a,b,c,d,e,f)
% Creation of the basic object SYS
% with 1 equation, 2 unknowns,    'LQ' type.
sys = SYS(2);
% Creation of the structure of data visible by the methods
% embedded in the class
obj.a = a;
obj.b = b;
obj.c = c;
obj.d = d;
obj.e = e;
obj.f = f;
% Creation of class 'QUADMINI' with the structure of data obj,
% and deriving from class SYS
obj = class(obj, 'QUADMINI', sys);
```

File : L0.m

```
function L0 = L0(obj)
L0 = obj.a;
```

File : L.m

```
function L = L(obj,U)
L = obj.b * U(1) + obj.c * U(2);
```

File : Q.m

```
function Q = Q(obj,U1,U2)
Q = obj.d * U1(1) * U2(2) + obj.e * U1(1) * U2(1) + obj.f * U1(2) * U2(2);
```

4.1.3 Launching the continuation

```
> manlabinit;
> ML_problem = QUADMINI(0,2,-2.3,1,2,3.2);
> ML_Ustart = [0;0];
> ML_dispvars = [1,2];
> manlabstart;
```

4.2 EXAMPLE WITH BIFURCATION POINTS

4.2.1 Problem statement

Lets consider the following problem with a single equation and two unknowns:

$$R(U) = R([x, y]^T) = (x - y^2)(y - (x - 2)^2) + a \tag{4.4}$$

where a is a constant parameter. Thanks to the new variable $v = x - y^2$ and $w = y - (x - 2)^2$, $R(U)$ can be rewritten as:

$$R(U) = R([x, y, v, w]^T) = \begin{cases} vw + a & = 0 \\ x - v - y^2 & = 0 \\ 4 - 4x + w - y + x^2 & = 0 \end{cases} \tag{4.5}$$

which can be recasted quadratically

$$R(U) = L0 + L(U) + Q(U, U) \tag{4.6}$$

with

$$L0 = \begin{cases} a \\ 0 \\ 4 \end{cases}, \quad L(U) = \begin{cases} 0 \\ x - v \\ -4x + w - y \end{cases}, \quad Q(U1, U2) = \begin{cases} v_1 w_2 \\ -y_1 y_2 \\ x_1 x_2 \end{cases} \tag{4.7}$$

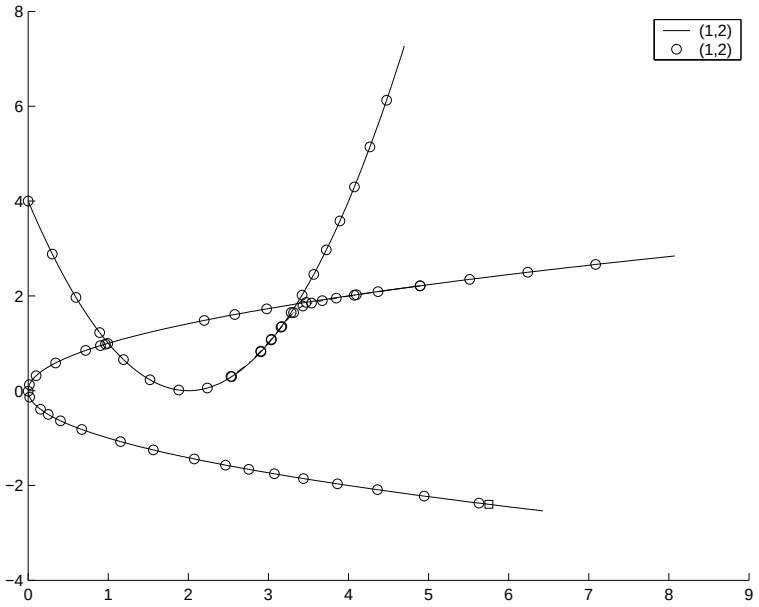


Figure 4.2: *Continuation diagram of the QUADBIF problem, composed of two parabolas intersecting at two bifurcation points.*

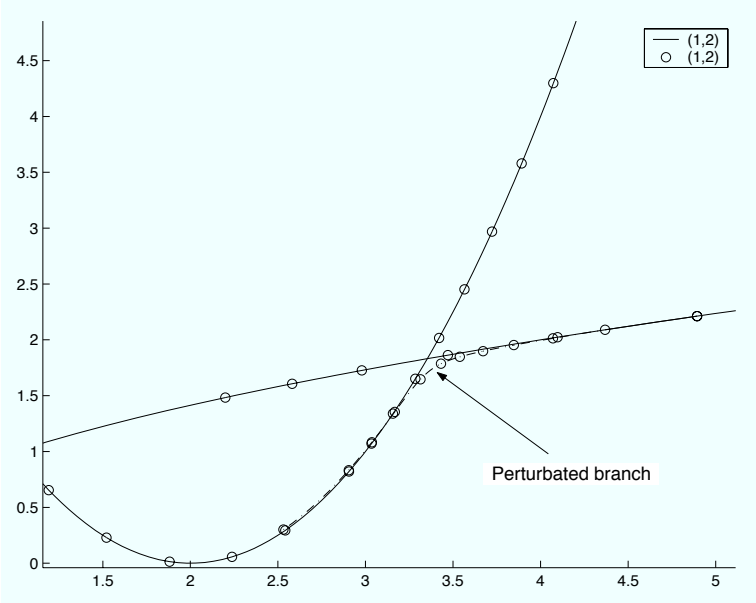


Figure 4.3: *Zoom on the continuation diagram of the QUADBIF problem around one bifurcation point. The use of a perturbated branch allowed the switch between one parabola and the other.*

4.2.2 Definition of the user problem

Here are 4 files allowing to perform the continuation of the solutions with **Manlab**

File : QUADBIF.m

```
function obj = QUADBIF(a)
% Creation of the basic object SYS
% with 3 equations, 4 unknowns, 'LQ' type.
sys = SYS(4);
% Creation of the structure of data visible by the methods
% embedded in the class
obj.a = a;
% Creation of class 'QUADBIF' with the structure of data obj,
% and deriving from class SYS
obj = class(obj, 'QUADBIF', sys);
% U = [x, y , v, w]
```

File : L0.m

```
function L0 = L0(obj)
L0 = zeros(3,1);
L0(1) = obj.a;
L0(2) = 0;
L0(3) = 4;
```

File : L.m

```
function L = L(obj,U)
L = zeros(3,1);
L(1) = 0;
L(2) = U(1)-U(3);
L(3) = -4*U(1) + U(4) - U(2) ;
```

File : Q.m

```
function Q = Q(obj,U1,U2)
Q = zeros(3,1);
```

```
Q(1) = U1(3) * U2(4);
Q(2) = -U1(2) * U2(2);
Q(3) = U1(1) * U2(1);
```

4.2.3 Launching the continuation

```
> manlabinit;
> ML_problem = QUADBIF(0);
% changing the order of the serie expansion (ANM):
> ML_problem = set_ordre(ML_problem, 25);
> ML_Ustart = [2;sqrt(2);0;0];
> ML_dispvars = [1,2];
> manlabstart;
```

4.3 A MECHANICAL EXAMPLE : BARRES

MANLAB/EXAMPLES/BARRES/

This is the 2 bar system in compression as described in [8].

- u and v are the displacements of the point of loading
- λ is the load parameter
- we use the additional variable : $w = -4u + u^2 + v^2$

Equilibrium equations:

$$w(u-2) = lambda$$

$$2v + vw = 0$$

The unknowns are : $U = [u\ v\ w\ \lambda]$

Launching the example: Type `lance` to launch the example.

4.4 A CHEMICAL REACTION EXAMPLE : BRUSSELATOR

`MANLAB/EXAMPLES/BRUSSELATOR/`

A discretised version of the chemical reaction model known as BRUSSELATOR. References : see Seydel's book [2], chapter 5 page 188.

Initial equations:

$$
\begin{aligned}
2-7u1 &\quad +u1u1u2+\lambda(u3-u1) &&= 0\\
6u1 &\quad -u1u1u2+10\lambda(u4-u2) &&= 0\\
2-7u3 &\quad +u3u3u4+\lambda(u1+u5-2u3) &&= 0\\
6u3 &\quad -u3u3u4+10\lambda(u2+u6-2u4) &&= 0\\
2-7u5 &\quad +u5u5u6+\lambda(u3-u5) &&= 0\\
6u5 &\quad -u5u5u6+10\lambda(u4+u6) &&= 0
\end{aligned}
$$

Easily put under quadratic form using the additional variables :

$$u7 = u1u1$$

$$u8 = u3u3$$

$$u9 = u5u5$$

It is then a problem with 9 equations and 10 unknowns : $U = [u1\ u2\ ....\ u9\ lambda]$

Launching the example: To launch the example, just go to the right directory and type `lance` in the **Matlab** command line.

4.5 EXAMPLE OF AN ELECTRO-CHEMICAL REACTION

`MANLAB/EXAMPLES/ECrea/`

Not documented yet.

4.6 BUCKLING INSTABILITY

`MANLAB/EXAMPLES/STATIC/DUFFING_FLAMBAGE_STAB/`

Buckling of a DUFFING like equation:

Initial equation: Duffing equation with a buckling parameter lambda writes:

$$u'' + \mu u' + (\omega_0^2 - \lambda)u + \Gamma u^3 = 0$$

with :

- u an unknown function of time
- λ a bufurcation parameter (real parameter)
- ω_0 the natural frequency (real parameter)
- μ the damping coefficient (real parameter)
- Γ the non-linearity coefficient

First order dynamical system: We define $v = u'$ to obtain:

$$\begin{aligned} u' &= +v \\ v' &= -\omega_0^2 u - \mu v + \lambda u - \Gamma u^3 \end{aligned}$$

Quadratic recast: We define $w = u^2$ and write:

$$\begin{aligned} u' &= v \\ v' &= -\omega_0^2 u - \mu v + \lambda u - \Gamma u w \\ 0 &= +w - u^2 \end{aligned}$$

In statics, u'=v=0 and v'=0 so that:

$$\begin{aligned} 0 = 0 &\; -\omega_0^2 u + \lambda u - \Gamma u w \\ 0 = 0 &\; +w - u^2 \end{aligned}$$

is a quadratic algebraic system of the form $L0(U) + L(U) + Q(U, U) = 0$ with $U = [u\ w\ \lambda]$.

The Jacobian of the initial dynamical system is:

$$JT = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 + \lambda - 3\Gamma u^2 & -mu \end{bmatrix}.$$

Launching the example: To launch the example, edit the `lance.m` script and correct the path to your **Manlab** directory, then type `lance` in the **Matlab** command line.

Chapter 5

Advanced Examples I : periodic orbits using HBM

The examples of this chapter show the path-following of periodic orbits of non-linear dynamical systems using high-order harmonic balance. The reference article for the method of combining HBM with **Manlab** ANM continuation is [9]. You will find these examples in the directory:
`MANLAB/EXAMPLES/HBM/`

5.1 VAN DER POL OSCILLATOR

`MANLAB/EXAMPLES/HBM/VANDERPOL/`

This example is detailed and commented in [9].

To launch the example, check the `lance.m` script for correct path definition and then type `lance` in the **Matlab** command line.

5.2 THE ROSSLER MODEL

`MANLAB/EXAMPLES/HBM/ROSSLER/`

Model equation:

$$x' = -y - z$$
$$y' = x + a * y$$
$$z' = b + z(x - \lambda)$$

$x(t)$, $y(t)$, $z(t)$ are unknown functions. a,b,c real parameters.

Quadratic recast:

$$\begin{aligned} x' = &\quad -y - z \\ y' = &\quad x + a * y \\ z' = &\ b \qquad\qquad -lambda * z \ + z * x \end{aligned}$$

With $Z(t) = [x\ y\ z]$ (Neq=3) the system becomes $m(Z') = c0 + l0(Z) + \lambda l1(Z) + q(Z, Z)$ and is then treated with HBM.

Launching the example: To launch the example, check the path definition in the `lance.m` script and then type `lance` in the **Matlab** command line.

5.3 PHYSICAL MODEL OF A CLARINET

`MANLAB/EXAMPLES/HBM/CLARINETTE/`

This model is taken from the work of Silva et al. presented in [10].

Model equations:

$$x'' + q_r\omega_r x' + \omega_r^2 x = \omega_r^2 p \tag{5.1}$$

$$p_n'' + 2\alpha_n c p_n' + \omega_n^2 p_n = 2c/Lu' \quad \forall n = 1..N_m \tag{5.2}$$

$$u = \zeta(1 - \gamma + x)\sqrt{(}\gamma - p) \tag{5.3}$$

$$p = p_1 + ... + p_N \tag{5.4}$$

The unknowns are $x(t)$, $p_n(t)$ and their derivatives, $u(t)$

Real parameter : q_r, ω_r, α_n, c, ω_n, L, ζ

Number of acoustical modes : N_m

Bifurcation parameter : γ

1st order DS and quadratic recast: We set $y = x'$, $z_n = p_n'$ and $v = \sqrt{(}\gamma - p)$ then:

$$\begin{aligned}
x' &= y \\
y' &= \omega_r^2(p_1 + ... + p_N) - q_r\omega_r y - \omega_r^2 x \\
p_n' &= z_n && \text{for } n = 1..N_m \\
z_n' - 2c/Lu' &= -2\alpha_n c z_n - \omega_n^2 p_n && \text{for } n = 1..N_m \\
0 &= -u + \zeta(1 - \gamma + x)v \\
0 &= -v^2 + \gamma - p
\end{aligned}$$

With $Z(t) = [x\ y\ p1\ ...\ pn\ z1\ ...zn\ u\ v]$ ($Neq = 2 * N_m + 4$) the system becomes : $m(Z') = \lambda c1 + l0(Z) + \lambda l1(Z) + q(Z, Z)$. It is then treated using HBM.

Launching the example: To launch the example, first check the path definition in `lance.m` and then type `lance` in the **Matlab** command line.

Chapter 6

Advanced Examples II : stability of periodic orbits using fortran acceleration

In this chapter we describe examples that use HBM for periodic orbits continuation with stability analysis. Some examples use fortran acceleration.

6.1 FORCED DUFFING OSCILLATOR

`MANLAB/EXAMPLES/HBM/FORCED_DUFFING/`

This example uses fortran acceleration and stability analysis.

Initial equation: Duffing equation reads :

$$u'' + \mu u' + \omega_0^2 u + \Gamma u^3 = F cos(\lambda t)$$

where $u(t)$ is the unknown function and λ a real-valued parameter.

1st order DS and quadratic recast: Using $v = u'$ and $w = u^2$, we rewrite the original equation as:

$$\begin{aligned} u' &= 0 & +v & \\ v' &= FFcos(lambdat) & -ome0^2u - muv & -Gamma * uw \\ 0 &= 0 & +w & -u^2 \end{aligned}$$

This system is then treated using the HBM, as described in [9].

Launching the example: To launch the example, check the paths definitions in `lance.m` and `@DUFF1HB/Makefile` (if you are using Windows, copy instead the file `Makefile.win` from the `MANLAB\TEMPLATES\` directory), and then type `lance` in the **Matlab** command line.

6.2 FORCED DUFFING OSCILLATOR WITH PARAMETRIC EXCITATION

`MANLAB/EXAMPLES/HBM/PARAMEXC_DUFFING/`

This example does not use the Fortran acceleration.

Initial dynamical system:

$$u'' + \mu u' + \omega_0^2 u + \Gamma u^3 + \delta(ucos(2\Omega t)) = Fcos\Omega t$$

where $u(t)$ is a real unknown time function. The other parameters are real.

First order dynamical system: We introduce $v = u'$ to transform the initial equation into the first order dynamical system:

$$\begin{aligned} u' &= v \\ v' &= -\omega_0^2 u - \mu v - \Gamma u^3 + F cos(\Omega t) + \delta u cos(2\Omega t) \end{aligned}$$

The Jacobian writes:

$$J(t) = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 - 3\Gamma u^2 + \delta cos(2\Omega t) & -\mu \end{bmatrix}.$$

Quadratic recast: We then introduce $v = u'$, $w = u^2$ and $x = cos(2\Omega t)$ to transform the previous system into the first order system with quadratic non-linearities:

$$\begin{aligned} u' &= & v & & \\ v' &= F cos(\Omega t) & -\mu v - \omega_0^2 u & -\Gamma u w - \delta u x \\ 0 &= & -w & +u^2 & \\ 0 &= cos(2\Omega t) & -x & & \end{aligned}$$

with N_{eq}=4 equations.

The Jacobian now writes:

$$J(t) = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 - 3\Gamma u^2 + \delta x & -\mu \end{bmatrix}.$$

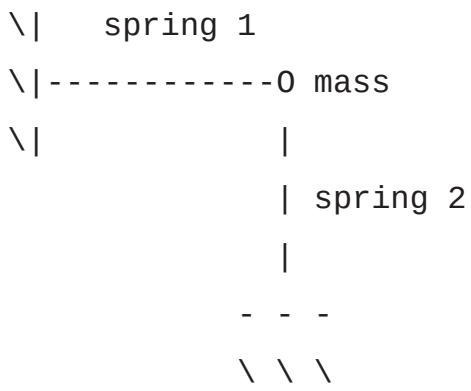
Launching the example: To launch the example, check the path definition in `lance.m` and then type `lance` in the **Matlab** command line.

6.3 NONLINEAR MODES OF A TWO-SPRING, ONE-MASS SYSTEM

`MANLAB/EXAMPLES/HBM/DEUX_RESSORTS/`

This example uses stability analysis and fortran acceleration.

Schematics of the model:



Notations:

- $u_1(t)$: x displacement of the mass
- $u_2(t)$: y displacement of the mass
- $e_i = u_i + 1/2(u_1^2 + u_2^2)$: strain for spring i (Green-lagrange)
- $N_i = k_i e_i$: spring force (k_i : stiffness)
- $W = \frac{1}{2}k_1 e_1^2 + \frac{1}{2}k_2 e_2^2$: strain energy
- m mass (m=1)

Model equations: The two governing equations are :

$$mu_1'' + \frac{\partial W}{\partial u_1} = 0$$
$$mu_2'' + \frac{\partial W}{\partial u_2} = 0$$

With the given notations, it becomes :

$$mu_1'' + N_1(1 + u_1) + N_2(u_1) = 0$$
$$mu_2'' + N_1(u_2) + N_2(1 + u_2) = 0$$

The system is autonomous and conservative. It has a first integral corresponding to the conservation of the total energy. It is quite different from dissipative systems, in terms of periodic solutions. In dissipative systems, periodic solution are generically isolated and an external parameter is required in order to continue the family of periodic orbits (see Van der Pol, for example).

In conservative (Hamiltonian) system, periodic orbit generally belongs to one-dimensional family of periodic solution, parametrised by the value of the first integral (here, the total energy). These one-dimensional families of periodic orbits are the non linear modes of the system.

It is convenient to have the same framework and the same software to find the branch of periodic solution of dissipative system and the non linear modes of conservative ones. To put the conservative system into the usual framework (dissipative with one parameter) we add dissipative terms $\lambda u_1'$ and $\lambda u2'$ in each equation :

$$mu_1'' + \lambda u_1' + N_1(1 + u_1) + N_2(u_1) = 0$$
$$mu_2'' + \lambda u_2' + N_1(u_2) + N_2(1 + u_2) = 0$$

This new system is dissipative and it has a free parameter, λ. It is easy to show that periodic solution can occur only when lambda is zero. So, this dissipative (with a parameter) has the same periodic solution as the conservative one (see Munoz-Almaraz Freire Galan Doedel Vanderbauwhede "Continuation of periodic orbits in conservative and Hamiltonian system", Physica D, 181, 1-38, 2003)

Quadratic recast: We transform into first order system by introducing $v_1 = u_1'$ and $v_2 = u_2'$.

Finally we have a system of $Neq = 6$ equations (ODEs and AEs) with quadratic

polynomial NL :

$$\begin{aligned} u_1' &= v_1 \\ u_2' &= v_2 \\ v_1' &= -\tfrac{1}{m}(N_1 \quad +\lambda u_1' \; +u_1(N_1+N_2)) \\ v_2' &= -\tfrac{1}{m}(N_2 \quad +\lambda u_2' \; +u_2(N_1+N_2)) \\ 0 &= N_1 - k_1 u_1 \qquad -\tfrac{1}{2}k_1(u_1^2+u_2^2) \\ 0 &= N_2 - k_2 u_2 \qquad -\tfrac{1}{2}k_2(u_1^2+u_2^2) \end{aligned}$$

This system is then treated using HBM.

Launching the example: To launch the example, check the paths definitions in `lance.m` and `@DEUXRES/Makefile` (if you are using Windows, copy instead the file `Makefile.win` from the `MANLAB\TEMPLATES\` directory), and then type `lance` in the **Matlab** command line.

6.4 FREE DUFFING OSCILLATOR

`MANLAB/EXAMPLES/HBM/FREE_DUFFING/`

This example uses stability analysis and fortran acceleration.

Initial dynamical system:

$$u'' + \lambda u' + \omega_0^2 u + \Gamma u^3 = 0$$

where $u(t)$ is a real unknown time function. The other parameters are real.

First order dynamical system and quadratic recast: We introduce $v = u'$ and $w = u^2$ to transform the previous equation into:

$$\begin{aligned} u' &= v \\ v' &= -\omega_0^2 u - \lambda v \; -\Gamma u^3 \\ 0 &= w \qquad\qquad -u^2 \end{aligned}$$

with $Neq = 3$ equations.

The system is then treated using HBM : periodic orbits arise when $\lambda \to 0$.

The Jacobian used for stability analysis is written:

$$J(t) = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 - 3\Gamma u^2 & -\mu \end{bmatrix}.$$

Launching the example: To launch the example, check the paths definitions in `lance.m` and `@DUFF1HB/Makefile` (if you are using Windows, copy instead the file `Makefile.win` from the `MANLAB\TEMPLATES\` directory), and then type `lance` in the **Matlab** command line.

6.5 FREE DUFFING OSCILLATOR WITH ESSENTIAL N.L.

`MANLAB/EXAMPLES/HBM/FREE_DUFFING_ESSENTIAL/`

This example uses stability analysis and fortran acceleration.

This example uses the same equation as the previous one, but in the special case of $\omega_0 = 0$.

See previous example for details.

6.6 A 2:1 INTERNAL RESONANCE SYSTEM

`MANLAB/EXAMPLES/HBM/RESONANCE_1_2/`

This example uses stability analysis but not the fortran acceleration.

The system is a 2DOF oscillator that possesses a 2:1 internal resonance. See references : [11] and [12].

Initial dynamical system

$$u_1'' + \mu_1 u' + \omega_1^2 u_1 + \beta_1 u_1 u_2 = F cos(\omega t) u_2'' + \mu_2 u' + \omega_2^2 u_2 + \beta_2 u_1^2 = 0$$

$u_1(t)$, $u_2(t)$ are real unknown time functions. The other parameters are real.

First order dynamical system We introduce $v1 = u1'$ and $v2 = u2'$ to transform the previous system into the first order, quadratic polynomial, dynamical system:

$$\begin{array}{lllll} u_1' = & & +v_1 & \\ u_2' = & & +v_2 & \\ v_1' = & F cos(\omega t) & -\omega_1^2 u_1 - \mu_1 v_1 & -\beta_1 u_1 u_2 \\ v_2' = & 0 & -\omega_2^2 u_2 - \mu_2 v_2 & -\beta_2 u_1^2. \end{array}$$

The Jacobian writes:

$$J = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\omega_1^2 - \beta_1 u_2 & -\beta_1 u_1 & -\mu_1 & 0 \\ -2\beta_2 u_1 & -\omega_2^2 & 0 & -\mu_2 \end{bmatrix}.$$

or, in the required form for Hill's method:

$$J0 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\omega_1^2 & 0 & -\mu_1 & 0 \\ 0 & -\omega_2^2 & 0 & -\mu_2 \end{bmatrix}$$

$$JL = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -\beta_1 u_2 & -\beta_1 u_1 & 0 & 0 \\ -2\beta_2 u_1 & 0 & 0 & 0 \end{bmatrix}$$

$$JQ = 0.$$

Launching the example Three scripts allow to launch the example :

- `lance_omeg1.m` : low-frequency excitation at ω. Two Neimark-Sacker (NS) bifurcations are found for $\omega = 0.9895$ and $\omega = 1.008$.
- `lance_omeg2_0m.m` : high-frequency excitation at ω. Two period-doubling (PD) bifurcations are found.
- `lance_omeg2.20m.m` : high-frequency excitation at 2ω. Enables to continue the branches from the PD.

Edit each launching script and check the path definition before executing by typing its name in the **Matlab** command line.

6.7 NONLINEAR NORMAL MODES OF A 2DOF SYSTEM

`MANLAB/EXAMPLES/NNM_2DDL_CUBIC`

This example uses stability analysis and fortran acceleration.

The system of this example is discussed in [13].

Initial dynamical system:

$$u_1'' + \lambda u_1 + 2u_1 - u_2 + 0.5u_1^3 = 0$$
$$u_2'' + \lambda u_2 + 2u_2 - u_1 = 0$$

$u_1(t)$ and $u_2(t)$ are two real unknown time functions. The other parameters are real.

First order dynamical system: We introduce $v_1 = u_1'$ and $v_2 = u_2'$ to transform the initial system into:

$$\begin{aligned} u_1' &= v_1 \\ u_2' &= v_2 \\ v_1' &= -2u_1 + u_2 \ -\lambda v_1 \ -\tfrac{1}{2}u_1^3 \\ v_2' &= -2u_2 + u_1 \ -\lambda v_2. \end{aligned}$$

The Jacobian writes:

$$J(t) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 - \frac{3}{2}u_1^2 & 1 & -\lambda 0 & \\ 1 & -2 & 0 & -\lambda \end{bmatrix}.$$

Quadratic dynamical system: We introduce $w = u_1^2$ to transform the previous system into:

$$\begin{aligned} u_1' &= v_1 \\ u_2' &= v_2 \\ v_1' &= -2u_1 + u_2 \ -\lambda v_1 \ -\tfrac{1}{2}u_1^3 \\ v_2' &= -2u_2 + u_1 \ -\lambda v_2. \\ 0 &= w \qquad\qquad -u_1^2. \end{aligned}$$

The system is then treated using HBM.

Launching the example: There are two scripts to launch the example:

- `lance.m` : standard system
- `lanceNRJ.m` : standard system with additional energy equation

To launch the example, check the paths definitions in each script and in the corresponding `Makefile` (if you are using Windows, copy instead the file `Makefile.win` from the `MANLAB\TEMPLATES\` directory), and then execute it by typing its name in the **Matlab** command line.

Bibliography

[1] E.J. Doedel, H. Keller, and J. Kernevez. Numerical analysis and control of bifurcation problems (i) bifurcation in finite dimension. *International journal of bifurcation and chaos*, 1:493–520, 1991.

[2] R. Seydel. *Practical Bifurcation and Stability Analysis, from equilibrium to chaos*. Springer-Verlag, second edition, 1994.

[3] B. Cochelin, N. Damil, and M. Potier-Ferry. Asymptotic numerical methods and pade approximants for non-linear elastic structures. *International journal for numerical methods in engineering*, 37:1187–1213, 1994.

[4] B. Cochelin, N. Damil, and M. Potier-Ferry. *Méthode asymptotique numérique*. Hermes Lavoisier, 2007.

[5] K. Georg and E.L. Allgower. Numerical continuation method, an introduction. In *Springer Series in Comptutational Mathematics*, volume 13. Springer-Verlag, 1990.

[6] E.J. Doedel. *Numerical Continuation methods for dynamical systems*, chapter Lecture notes on numerical analysis of nonlinear equations, pages 1–49. B. Krauskopf H.M. Osinga J. Galan-Vioque Eds, Springer Verlag, 2007.

[7] Arnaud Lazarus and Olivier Thomas. A harmonic-based method for computing the stability of periodic solutions of dynamical systems. *Comptes Rendus Mécanique*, 338(9):510 – 517, 2010.

[8] S. Baguet and B. Cochelin. On the behaviour of the anm continuation in the presence of bifurcations. *Communications in numerical methods in engineering*, 19:459–471, 2003.

[9] Bruno Cochelin and Christophe Vergez. A high order purely frequency-based harmonic balance formulation for continuation of periodic solutions. *Journal of Sound and Vibration*, 324:243–262, 2009.

[10] Fabrice Silva, Vincent Debut, Jean Kergomard, Christophe Vergez, Aude Lizée Deblevid, and Philippe Guillemain. Simulation of single reed instruments oscillations based on modal decomposition of bore and reed dynamics. In *Proceedings of the International Congress of Acoustics*, Madrid, Spain, September 2007.

[11] A.H. Nayfeh and D.T. Mook. *Nonlinear Oscillations*. Wiley, 1979.

[12] O. Thomas, C. Touze, and A. Chaigne. Non-linear vibrations of free-edge thin spherical shells: modal interaction rules and 1:1:2 internal resonance. *International Journal of Solids and Structures*, 42:3339–3373, 2005.

[13] G. Kerschen, M. Peeters, J.C. Golinval, and A.F. Vakakis. Nonlinear normal modes, part i: A useful framework for the structural dynamicist. *Mechanical Systems and Signal Processing*, 23(1):170 – 194, 2009. Special Issue: Nonlinear Structural Dynamics.